Time Complexity Bounds for Shared-memory Mutual Exclusion

by
Yong-Jik Kim

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2003

Approved by:

James H. Anderson, Advisor

Sanjoy K. Baruah, Reader

Jan F. Prins, Reader

Michael Merritt, Reader

Lars S. Nyland, Reader

Jack Snoeyink, Reader

ABSTRACT
YONG-JIK KIM: Time Complexity Bounds for Shared-memory Mutual Exclusion.
(Under the direction of James H. Anderson.)

Mutual exclusion algorithms are used to resolve conflicting accesses to shared resources by concurrent processes. The problem of designing such an algorithm is widely regarded as one of the "classic" problems in concurrent programming.

Recent work on scalable shared-memory mutual exclusion algorithms has shown that *the* most crucial factor in determining an algorithm's performance is the amount of traffic it generates on the processors-to-memory interconnect [23, 38, 61, 84]. In light of this, the *RMR* (*remote-memory-reference*) *time complexity* measure was proposed [84]. Under this measure, an algorithm's time complexity is defined to be the worst-case number of remote memory references required by one process in order to enter and then exit its critical section.

In the study of shared-memory mutual exclusion algorithms, the following fundamental question arises: *for a given system model, what is the most efficient mutual exclusion algorithm that can be designed under the RMR measure?* This question is important because its answer enables us to compare the cost of synchronization in different systems with greater accuracy. With some primitives, constant-time algorithms are known, so the answer to this question is trivial. However, with other primitives (*e.g.*, under read/write atomicity), there are still gaps between the best known algorithms and lower bounds.

In this dissertation, we address this question. The main thesis to be supported by this dissertation can be summarized as follows. *The mutual exclusion problem exhibits different time-complexity bounds as a function of both the number of processes* (N) *and the number of simultaneously active processes* (*contention*, k), *depending on the available synchronization primitives and the underlying system model. Moreover, these time-complexity bounds are nontrivial, in that constant-time algorithms are impossible in many cases.*

In support of this thesis, we present a time-complexity lower bound of $\Omega(\log N/\log\log N)$ for systems using atomic reads, writes, and comparison primitives, such as *compare-and-swap*. This bound is within a factor of $\Theta(\log\log N)$ of being optimal.

Given that constant-time algorithms based on *fetch-and-ϕ* primitives exist, this lower bound points to an unexpected weakness of *compare-and-swap*, which is widely regarded as being the most useful of all primitives to provide in hardware.

We also present an adaptive algorithm with $\Theta(\min(k, \log N))$ RMR time complexity under read/write atomicity, where k is contention. In addition, we present another lower bound that precludes the possibility of an $o(k)$ algorithm for such systems, even if comparison primitives are allowed.

Regarding nonatomic systems, we present a $\Theta(\log N)$ nonatomic algorithm, and show that adaptive mutual exclusion is impossible in such systems by proving that any nonatomic algorithm must have a single-process execution that accesses $\Omega(\log N/\log\log N)$ distinct variables.

Finally, we present a generic *fetch-and-ϕ*-based local-spin mutual exclusion algorithm with $\Theta(\log_r N)$ RMR time complexity. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank r, where $2 \leq r < N$. The *rank* of a *fetch-and-ϕ* primitive expresses the extent to which processes may "order themselves" using that primitive. For primitives that meet a certain additional condition, we present a $\Theta(\log N/\log\log N)$ algorithm, which is time-optimal for certain primitives of constant rank.

ACKNOWLEDGMENTS

First, I want to thank my advisor, Jim Anderson, for his support and enthusiasm. I have learned an enormous amount from working with him, and this dissertation would not have been possible without his far-reaching insight and constant encouragement.

I also want to thank Michael Merritt, who took the trouble of reading this large dissertation and flying to attend my defense. Thanks, too, to the rest of my committee: Jan Prins, Jack Snoeyink, Sanjoy Baruah, and Lars Nyland. I greatly appreciate their willingness to bend their schedules to accommodate countless meetings and exams, and also their support and encouragement throughout the process. I would also like to thank Guido Gerig, my ex-advisor, and people in the medical imaging group, for being supportive and friendly throughout my stay at UNC.

In addition, I would like to thank people in the distributed algorithms research community for lively discussions and reviewing earlier versions of the results presented in this dissertation. I cannot hope to recall everyone who should be mentioned, but the list would surely include the following people: Hagit Attiya, Faith Fich, Maurice Herlihy, Leslie Lamport, Victor Luchangco, Maged Michael, Mark Moir, Eric Ruppert, Jennifer Welch, and many others.

Many thanks to Anand Srinivasan, Phil Holman, and Shelby Funk, for their friendship and their help in term projects, writing papers, and other things. Lucia Cevidanes, I enjoyed working with you; I hope you will find a better matlab programmer who can decipher my code!

Special thanks to the Korean friends who made my stay in Chapel Hill a lot more homelike: Juhyun, Sungeui, Joohee (of the math department), Joohi, Sang-Uok and Hye-Chung, In-Kyung, Hyemi Choi, Kuan-Hui Lee, and (of course) many, many others.

No words will be enough to thank my family for their love and support. Thank you, Mom and Dad, Namhee, Heeyeon, and Yonggon. I love all of you.

Finally, I would like to thank Eunjung, my lovely wife. Throughout the years we have been together, she has always been kind, understanding, and inspiring. Without her, I would have missed much of what life is.

CONTENTS

LIST OF TABLES **x**

LIST OF FIGURES **xi**

1 Introduction **1**

1.1 Known Local-spin Algorithms 4

1.2 Known Adaptive Algorithms 5

1.3 Known Nonatomic Algorithms 6

1.4 Contributions 6

1.4.1 General Mutual Exclusion 8

1.4.2 Adaptive Mutual Exclusion 9

1.4.3 Nonatomic Mutual Exclusion 9

1.4.4 Mutual Exclusion with *fetch-and-ϕ* Primitives 10

1.5 Organization 11

2 Related Work **12**

2.1 Preliminaries 13

2.2 Local-spin Algorithms 16

2.2.1 Algorithms that use *fetch-and-ϕ* Primitives 17

2.2.2 Algorithms that use Only Reads and Writes 23

2.3 Fast Mutual Exclusion 26

2.3.1 ALGORITHM L: Lamport's Fast Mutual Exclusion Algorithm . . 27

2.3.2 Fast Mutual Exclusion with Local Spinning 29

2.4 Adaptive Algorithms 30

2.4.1 Adaptive Algorithms using Filters . . . 32

2.4.2 An Adaptive Bakery Algorithm . . . 36

2.4.3 Adaptive Mutual Exclusion with Infinitely Many Processes . . . 37

2.4.4 Local-spin Adaptive Algorithms . . . 38

2.5 Mutual Exclusion with Nonatomic Operations . . . 38

2.6 Time- and Space-complexity Lower Bounds . . . 41

2.6.1 Overview of Proof Methodologies . . . 41

2.6.2 Lower-bound Results . . . 42

3 Space-optimal Mutual Exclusion Under Read/Write Atomicity 47

3.1 Transformation of Algorithm YA-N . . . 48

3.2 Time Complexity . . . 55

3.3 Concluding Remarks . . . 57

4 Adaptive Mutual Exclusion Under Read/Write Atomicity 59

4.1 Adaptive Algorithm . . . 61

4.1.1 Related Algorithms and Key Ideas . . . 61

4.1.2 Algorithm A-U: Unbounded Algorithm . . . 65

4.1.3 Algorithm A-B: Bounded Algorithm . . . 73

4.1.4 Algorithm A-LS: Linear-space Algorithm . . . 75

4.2 Concluding Remarks . . . 83

5 Time-complexity Lower Bound for General Mutual Exclusion 85

5.1 Definitions . . . 87

5.1.1 Atomic Shared-memory Systems . . . 87

5.1.2 Mutual-exclusion Systems . . . 93

5.1.3 Cache-coherent Systems . . . 95

5.2 Lower-bound Proof Strategy . . . 99

5.2.1 Process Groups and Regular Computations . . . 99

5.2.2 Detailed Proof Overview . . . 103

5.3 Detailed Lower-bound Proof . . . 110

5.4 Constant-time Algorithm for LFCU Systems . . . 137

5.5 Concluding Remarks 139

6 Time-complexity Lower Bound for Adaptive Mutual Exclusion 142

6.1 Proof Strategy 143

6.2 Detailed Lower-bound Proof 148

6.3 Concluding Remarks 167

7 Algorithm and Time-complexity Lower Bound for Nonatomic Systems 169

7.1 Nonatomic Algorithm 172

7.2 Nonatomic Shared-memory Systems 175

7.3 Lower Bound: Proof Sketch 180

7.3.1 Brief Overview 181

7.3.2 Formal Definitions 190

7.3.3 Detailed Proof Overview 198

7.4 Concluding Remarks 208

8 Generic Algorithm for *fetch-and-ϕ* Primitives 209

8.1 Definitions 212

8.2 Constant-time Generic Algorithm 213

8.2.1 ALGORITHM G-CC: A Generic Algorithm for CC Machines . . 214

8.2.2 ALGORITHM G-DSM: A Generic Algorithm for DSM Machines 218

8.3 $\Theta(\log N/\log\log N)$ Algorithms 221

8.3.1 ALGORITHM T0: A Simple Tree Algorithm 222

8.3.2 ALGORITHM T: A Generic Tree Algorithm 230

8.4 Concluding Remarks 240

9 Conclusion 241

9.1 Summary 241

9.2 Future Work 243

A Correctness Proof for ALGORITHM A-LS in Chapter 4 247

A.1 List of Invariants 251

A.2 Proof of the Exclusion Property 256
A.3 Proof of Adaptivity under the RMR Measure 273

B Detailed Proofs of Lemmas 5.1–5.6 285

C Correctness Proof for ALGORITHM NA in Chapter 7 309

D Detailed Proof of the Time-complexity Lower Bound Presented in Chapter 7 320

E Correctness Proof for ALGORITHM G-CC in Section 8.2.1 360

E.1 List of Invariants 362
E.2 Proof of the Exclusion Property 365
E.3 Proof of Starvation-freedom 385

F Correctness Proof for ALGORITHM T in Section 8.3 392

F.1 List of Invariants 396
F.2 Proof of the Exclusion Property 400
F.3 Proof of Starvation-freedom 434

BIBLIOGRAPHY 446

LIST OF TABLES

1.1 Contributions of this dissertation: lower and upper bounds for various classes of mutual exclusion algorithms. 8

2.1 Comparison of known adaptive algorithms. 31

4.1 Comparison of known adaptive algorithms, extended. 60

LIST OF FIGURES

1.1 Illustration of DSM and CC machines. 3

2.1 ALGORITHM TA: Array-based queue lock using *fetch-and-increment* and *fetch-and-add*. 17

2.2 ALGORITHM MCS: List-based queue lock using *fetch-and-store* and *compare-and-swap*. 20

2.3 ALGORITHM YA-2: A two-process version of ALGORITHM YA. 24

2.4 **(a)** ALGORITHM L: Lamport's fast mutual exclusion algorithm. **(b)** The splitter element and **(c)** its implementation. 28

2.5 **(a)** The filter element and **(b)** its implementation. 33

2.6 **(a)** A leader-election algorithm using filters. **(b)** Unbounded version of Choy and Singh's algorithm [30]. 34

2.7 ALGORITHM AST: An adaptive bakery algorithm by Afek, Stupp, and Touitou [5]. 36

2.8 ALGORITHM JA: Two-process case. 39

3.1 Variable declarations. 48

3.2 ALGORITHM YA-N and ALGORITHM CC. 49

3.3 ALGORITHM LS and ALGORITHM F. 52

4.1 The splitter element and the code fragment that implements it. 61

4.2 **(a)** Renaming grid (depicted for $N = 5$). **(b)** Renaming tree. 62

4.3 Renaming tree and overflow tree. 64

4.4 Variables used in ALGORITHM A-U. 65

4.5 ALGORITHM A-U: Adaptive algorithm with unbounded memory. . . . 66

4.6 ALGORITHM A-B: Adaptive algorithm with $\Theta(N^2)$ space complexity. . 73

4.7 ALGORITHM A-LS: adaptive algorithm with $\Theta(N)$ space complexity. . 76

4.8 Location of a process p during its entry and exit section. 79

4.9 Deflection of a process p at splitter i. 80

5.1 Transition events of an atomic mutual exclusion system. 94

5.2 Process groups. 101

5.3 Induction steps. 104

5.4 Extending a regular computation. 105

5.5 Erasing strategy. 106

5.6 Roll-forward strategy. Part I. 107

5.7 Roll-forward strategy. Part II. 109

5.8 Algorithm for arranging the events of S so that information flow is sufficiently low. 128

5.9 Algorithm for constructing $H_1, H_2, \ldots, H_k$. 136

5.10 A mutual exclusion algorithm with $O(1)$ time complexity in LFCU systems.138

6.1 Erasing strategy. 146

6.2 Roll-forward strategy. 147

7.1 ALGORITHMS YA-N and NA. 173

7.2 Overlapping reads and writes of the same variable. 177

7.3 A possible solo computation by a process p. 180

7.4 Structure of a regular computation H after $m - 1$ induction steps. . . . 182

7.5 Extensions of H. 183

7.6 Covering strategy. 188

7.7 The use of reserve processes to "exchange" two processes before erasing. 189

7.8 The structure of a regular computation. 193

7.9 The single-writer case. 195

7.10 Construction of Y from Act(H). 199

7.11 The chain-erasing procedure. 201

7.12 An example of chain erasing. 202

7.12 An example of chain erasing, continued. 203

7.13 Construction of the "conflict graph" $\mathcal{G}$. 205

7.14 Erasing strategy. 207

8.1 ALGORITHM G-CC: Generic *fetch-and-ϕ*-based mutual exclusion algorithm for CC machines. 214

8.2 The structure of ALGORITHM G-CC. 215

8.3 ALGORITHM G-DSM: Generic *fetch-and-ϕ*-based mutual exclusion algorithm for DSM machines. 219

8.4 Definitions of *NodeType*, *AcquireNode*, and *ReleaseNode*. 222

8.5 ALGORITHM T0: A tree-structured algorithm using a *NodeType* object. 223

8.6 Arbitration tree of ALGORITHMS T0 and T. 224

8.7 Structure of a node used in Algorithm T0. 225

8.8 Variables used in ALGORITHM T. 232

8.9 ALGORITHM T. 233

8.9 ALGORITHM T, continued. 234

A.1 ALGORITHM A-LS with auxiliary variables added. (Continued on the next page.) 248

A.1 ALGORITHM A-LS with auxiliary variables added, continued. 249

C.1 Recursive version of ALGORITHM NA. 310

D.1 The chain-erasing procedure to erase processes in $K = \{p_1,\ p_2,\ \ldots,\ p_h\}$.335

E.1 ALGORITHM G-CC with auxiliary variables added. 361

F.1 ALGORITHM T with auxiliary variables added. 393

F.1 ALGORITHM T with auxiliary variables added, continued. 394

F.1 ALGORITHM T with auxiliary variables added, continued. 395

CHAPTER 1

Introduction

Mutual exclusion algorithms are used to resolve conflicting accesses to shared resources by concurrent processes. The problem of designing such algorithms is widely regarded as one of the "classic" problems in concurrent programming. In the mutual exclusion problem, a process accesses the resource to be managed by executing a "critical section" of code. Activities not involving the resource occur within a corresponding "noncritical section." Before and after executing its critical section, a process executes two other code fragments, called "entry" and "exit" sections, respectively. A process may halt within its noncritical section but not within its critical section. Furthermore, no variables (other than program counters) accessed within a process's entry or exit section may be accessed within its critical or noncritical section. The objective is to design the entry and exit sections so that the following requirements hold.

- **Exclusion:** At most one process can execute its critical section at any time.
- **Livelock-freedom:** If some process is in its entry section, then some process eventually executes its critical section.

Often, livelock-freedom is replaced by the following stronger property.

- **Starvation-freedom:** If some process is in its entry section, then *that* process eventually executes its critical section.

For either variant, livelock-freedom or starvation-freedom, the following is required: if a process is in its exit section, then it eventually enters its noncritical section (this property holds trivially for most algorithms).

Previous work on mutual exclusion developed algorithms for both shared-memory and message-passing systems. In this dissertation, we present several results that pertain to shared-memory algorithms.

Shared-memory mutual exclusion algorithms can be divided into two categories, based on whether they employ operating-system services. Due to the Exclusion property, when a process is in its critical section, all other processes in their entry sections must wait. If operating-system services that suspend the execution of a waiting process are not available, then such waiting must be done by *busy waiting*: each waiting process repeatedly tests a condition by accessing shared variables, until that condition is satisfied.

In this dissertation, we consider only busy-waiting algorithms that do not use operating-system services. Although almost all modern operating systems support multiprogramming, busy-waiting algorithms still have two important applications. First, if the expected waiting duration is shorter than the overhead induced by a context switch, then it is desirable to avoid operating system calls altogether. Thus, busy waiting is more efficient in this case. Second, in a multiprocessor system, a processor with a waiting process may have no other job to execute. In such a case, invoking operating-system services does not provide any advantage.

Among recent research on shared-memory mutual exclusion, work on "local-spin" algorithms has been one of the most significant trends. Most early shared-memory algorithms employ somewhat complicated busy-waiting loops in which many shared variables are read and written [34, 49]. Under contention, such busy-waiting loops generate excessive traffic on the processors-to-memory interconnect, resulting in poor performance.

Local-spin mutual exclusion algorithms avoid this problem by requiring all busy-waiting loops to be read-only loops in which one or more "spin variables" are repeatedly tested and no shared variables are written. Such spin variables must be *locally accessible*, *i.e.*, they can be accessed without causing an interconnect traversal. Two architectural paradigms have been considered in the literature that allow shared variables to be locally accessed: distributed shared-memory (DSM) machines and cache-coherent (CC) machines. Both are illustrated in Figure 1.1. In a DSM machine, each processor has its own memory module that can be accessed without traversing the global interconnect. On such a machine, a shared variable can be made locally accessible by storing it in a local memory module. In a CC machine, each processor has a private cache, and some hardware protocol is used to enforce cache consistency. As a result, writable shared

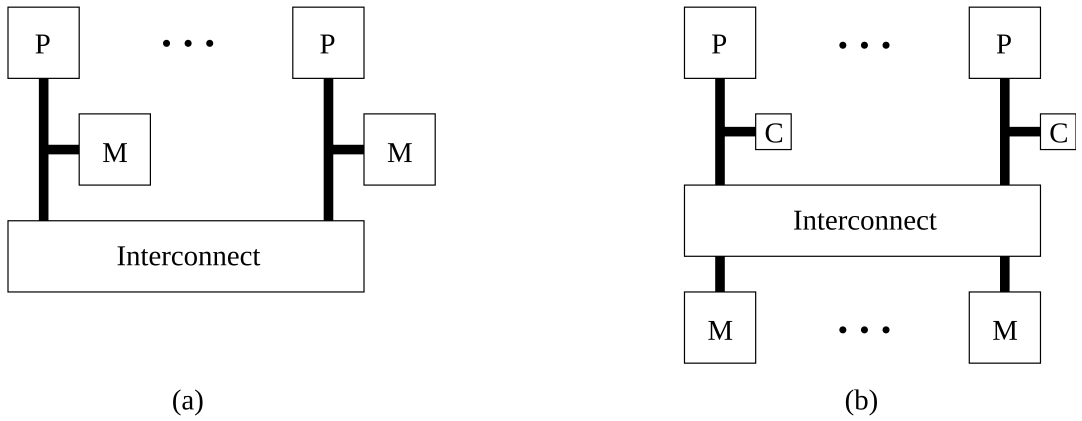


Figure 1.1: **(a)** DSM model. **(b)** CC model. In both insets, 'P' denotes a processor, 'C' a cache, and 'M' a memory module.

data can be cached. On such a machine, a shared variable becomes locally accessible by migrating to a local cache line. In this dissertation, we consider a DSM machine with caches that are kept coherent to be a CC machine.

We also assume that there is a unique process executing on each processor; we further assume that these processes do not migrate. If instead multiple processes are allowed to execute on the same processor (multiprogramming), then busy-waiting-based algorithms are probably inappropriate, since the preemption of a waiting process may allow better processor utilization. In addition, allowing process migration would complicate the definition of a "locally accessible" variable in a way that is unnecessarily distracting.

In local-spin algorithms for DSM machines, each process must have its own dedicated spin variables (which must be stored in its local memory module). In contrast, in algorithms for CC machines, processes may *share* spin variables, because each process can read a different cached copy. It is easy to see that any algorithm that locally spins on a DSM machine will also locally spin on a CC machine, while the reverse is not necessarily true. For this reason, it is generally more difficult to design local-spin algorithms for the DSM model. (Although virtually every modern multiprocessor is cache-coherent, non-cache-coherent DSM systems are still used in embedded applications, where cheaper computing technology often must be used due to cost limitations. Thus, the DSM model is of relevance for reasons other than historical interest.)

Recent work on scalable local-spin mutual exclusion algorithms has shown that *the* most crucial factor in determining an algorithm's performance is the amount of interconnect traffic it generates [23, 38, 61, 84]. In light of this, we adopt the *RMR* (*remote-memory-reference*) *time complexity* measure [84] throughout most of this dissertation: the RMR time complexity of a mutual exclusion algorithm is defined to be the worst-case number of remote memory references by one process in order to enter and then exit its critical section. A *remote memory reference* is a shared variable access that requires an interconnect traversal.

An algorithm may have different RMR time complexities under the CC and DSM models because the notion of a remote memory reference differs under these two models. In the CC model, we assume that, once a spin variable has been cached, it remains cached until it is either updated or invalidated as a result of being modified by another process on a different processor. In other words, we ignore any cache displacements caused by cache capacity or associativity constraints or by the execution of code within the operating system (*e.g.*, interrupt service routines).

Before describing the contributions of this dissertation, we first review relevant work on mutual exclusion in the following sections. In particular, we discuss known local-spin algorithms in Section 1.1, adaptive algorithms in Section 1.2, and nonatomic algorithms in Section 1.3.

1.1 Known Local-spin Algorithms

The first local-spin algorithms were algorithms in which *fetch-and-ϕ* primitives[1] are used to enqueue blocked processes onto the end of a "spin queue" [23, 38, 61]. In each of these algorithms, a constant number of remote memory references is required per critical-section execution. The algorithms vary in the synchronization primitives used, and whether spinning is local on both CC and DSM systems. The three most well-known such algorithms are due to T. Anderson [23], Graunke and Thakkar [38], and Mellor-Crummey and Scott [61].

- T. Anderson's queue-based algorithm [23] uses both *fetch-and-increment* and *fetch-and-add*. This algorithm has $O(1)$ RMR time complexity under the CC model.

- Graunke and Thakkar's algorithm [38] uses *fetch-and-store*. This algorithm has $O(1)$ RMR time complexity under the CC model.

[1]Formal definitions of the primitives mentioned in this section are given in Chapter 2.

- Mellor-Crummey and Scott's algorithm [61] uses both *compare-and-swap* and *fetch-and-store*. This algorithm has $O(1)$ RMR time complexity under both the DSM and CC model.

Each of the algorithms mentioned above requires one or more strong synchronization primitives. This led some researchers to question whether such primitives were in fact *necessary* for local-spin synchronization. Anderson presented the first local-spin mutual exclusion algorithm that uses only read and write operations. His algorithm has $\Theta(N)$ RMR time complexity, where N is the number of processes [11].[2] Later, Yang and Anderson presented a read/write algorithm with $\Theta(\log N)$ RMR time complexity, in which instances of a local-spin mutual exclusion algorithm for two processes are embedded within a binary arbitration tree [84].

1.2 Known Adaptive Algorithms

It is generally believed that "contention for a critical section is rare in well-designed systems" [56]. Thus, it is desirable that the time complexity of a mutual exclusion algorithm should not depend exclusively on N (the number of processes), which may be much larger than the number of concurrently active processes. This insight led to a major research trend in work on mutual exclusion, namely, the study of "adaptive" algorithms.

A mutual exclusion algorithm is *adaptive* if its time complexity (under some measure) is a function of the number of contending processes [30, 63, 78]. Adaptive mutual exclusion algorithms based on read and write operations have been proposed by Styer [78], Choy and Singh [30], Attiya and Bortnikov [24], and Afek, Stupp and Touitou [5]. Each of these algorithms is a non-local-spin algorithm,[3] and is adaptive under some time complexity measure. (A detailed discussion of various time complexity measures of relevance in work on adaptive algorithms is given in Section 2.4.)

[2]Throughout this dissertation, N is defined to be the total number of processes, and is assumed to be known *a priori*.

[3]Actually, these algorithms are adaptive under the RMR measure on CC machines with *write-update* caches. In a system with write-update caches, when a processor writes to a variable v that is also cached on other processors, a message is sent to these processors so that they can *update* v's value and maintain cache consistency.

Since these algorithms are quite complicated, it is unclear whether they are adaptive on CC machines in general.

1.3 Known Nonatomic Algorithms

In most of the algorithms mentioned so far, shared variables are assumed to be accessed atomically. However, requiring atomic memory access is tantamount to assuming mutual exclusion in hardware [51]. Thus, mutual exclusion algorithms requiring this are in some sense circular.

In *nonatomic* algorithms, variable accesses are assumed to take place over intervals of time, and hence may overlap one another. Lamport presented the first nonatomic algorithm, his famous bakery algorithm [51], which is not a local-spin algorithm and uses variables of unbounded range. In later work, Lamport presented four other nonatomic algorithms, each with bounded memory [54]. These algorithms differ in the progress and fault-tolerance properties they satisfy. None are local-spin algorithms. Anderson's $\Theta(N)$ local-spin algorithm [11], which is mentioned in Section 1.1, was also the first nonatomic local-spin algorithm.

1.4 Contributions

Recent work on shared-memory mutual exclusion has shed some light on the following fundamental question: for a given system model, what is the most efficient mutual exclusion algorithm that can be designed under the RMR measure? This question is important because its answer enables us to compare the cost of synchronization in different systems with greater accuracy. For example, its answer may help system designers to analyze the relative costs and merits of different synchronization primitives that may be provided.

With primitives such as *fetch-and-increment* and *fetch-and-store*, the answer to the question above is trivial, because constant-time algorithms are already known [23, 38, 61], as mentioned in Section 1.1. However, with other primitives, there are still gaps between the best known algorithms and lower bounds. For example, under read/write atomicity, Yang and Anderson's algorithm [84], the most efficient algorithm currently known, has $\Theta(\log N)$ time complexity.

Anderson and Yang [22] presented several lower bounds that establish trade-offs between the RMR time complexity required for mutual exclusion and write- and access-contention, where *write-contention* (*access-contention*) is the number of processes that may potentially be simultaneously enabled to write (access) the same shared variable.

Cypher [33] was the first to present a lower bound under the RMR measure under arbitrary access-contention. In particular, he established a lower bound of $\Omega(\log\log N/\log\log\log N)$ for systems with reads, writes, and comparison primitives. In this dissertation, we present a substantially improved lower bound of $\Omega(\log N/\log\log N)$ in Chapter 5.

These results indicate that we can obtain a nontrivial classification of (shared-memory) system models based on the time complexity of available mutual exclusion algorithms. Moreover, as shown shortly, this classification extends (in a nontrivial way) to a wide variety of models and algorithms, such as nonatomic and adaptive algorithms.

The main thesis to be supported by the work in this dissertation is directed at the question above. This thesis is stated below.

The mutual exclusion problem exhibits different time complexity bounds as a function of both the number of processes (N) and current contention (k), depending on the available synchronization primitives.

- *In a nonatomic system, $\Omega(\log N/\log\log N)$ RMR time complexity is required of any mutual exclusion algorithm, and thus adaptive algorithms are impossible.*
- *In an atomic system, $O(\min(k, \max(1, \log_r N)))$ RMR time complexity is possible, if a primitive of "rank" r is available (the notion of a rank is introduced later in this dissertation). Moreover, in a system with only reads, writes, and comparison primitives, $\Omega(\log N/\log\log N)$ RMR time complexity is required of any mutual exclusion algorithm, and $o(k)$ RMR time complexity is impossible.*

Many of the results presented in this dissertation support this thesis directly, while others are by-products of this research. In the following subsections, we describe the contributions of this dissertation in detail. Lower and upper time-complexity bounds for various classes of mutual exclusion algorithms are summarized in Table 1.1. In this table, inset (a) lists various results obtained before the research reported in this dissertation; inset (b) lists the contributions of this dissertation, which are described below.

Class of algorithms	**Upper bound**	**Lower bound**
General algorithms with reads, writes, and comparison primitives	$\Theta(\log N)$ [84]	$\Omega(\log \log N/ \log \log \log N)$ [33]
Adaptive algorithms with reads, writes, and comparison primitives	(no local-spin algorithm under the DSM model)	—
Nonatomic algorithms	$\Theta(N)$ [11]	—
Algorithms with *fetch-and-ϕ* primitives		
• *fetch-and-increment* and *fetch-and-add*	$\Theta(1)$ (CC model) [23]	$\Theta(1)$ (CC model)
• *fetch-and-store*	$\Theta(1)$ (CC model) [38]	$\Theta(1)$ (CC model)
• *fetch-and-increment* and *compare-and-swap*	$\Theta(1)$ [61]	$\Theta(1)$

(a) Results obtained before my research.

Class of algorithms	**Upper bound**	**Lower bound**
General algorithms with reads, writes, and comparison primitives	$\Theta(\log N)$ [84]	$\Omega(\log N/ \log \log N)$ (Ch. 5)
Adaptive algorithms with reads, writes, and comparison primitives	$O(\min(k, \log N))$ (Ch. 4)	$o(k)$ is impossible (Ch. 6)
Nonatomic algorithms	$\Theta(\log N)$ (Ch. 7)	$\Omega(\log N/ \log \log N)$ (Ch. 7)
Algorithms with *fetch-and-ϕ* primitives with rank r	$O(\min(k, \max(1, \log_r N)))$ (Ch. 8)	—

(b) Contributions of this dissertation.

Table 1.1: Contributions of this dissertation: lower and upper bounds for various classes of mutual exclusion algorithms. Each entry applies to both DSM and CC system models unless otherwise noted. In this table, N and k denote the number of processes and current contention, respectively.

1.4.1 General Mutual Exclusion

In Chapter 5, we present a time-complexity lower bound of $\Omega(\log N/ \log \log N)$ for systems using reads, writes, and comparison primitives, such as *compare-and-swap*. Given Yang and Anderson's algorithm [84], this bound is within a factor of $\Theta(\log \log N)$ of being optimal.

Given that constant-time algorithms based on *fetch-and-ϕ* primitives exist, this lower bound points to an unexpected weakness of *compare-and-swap*, which is widely regarded as being the most useful of all primitives to provide in hardware. In particular, this result implies that the *best* algorithm based on *compare-and-swap* can have RMR time complexity that is at most $\Theta(1/ \log \log N)$ times that of the best algorithm based on reads and writes.

1.4.2 Adaptive Mutual Exclusion

In Chapter 4, we present an adaptive algorithm with $\Theta(\min(k, \log N))$ RMR time complexity under read/write atomicity, where k is contention. This is the first adaptive local-spin algorithm ever proposed for this model. (As discussed later, Afek, Stupp, and Touitou [6] have independently devised another local-spin adaptive algorithm, with a structure similar to our algorithm.)

We also present another lower bound that precludes the possibility of an $o(k)$ algorithm for such systems. In particular, we prove the following:

> *For any k, there exists some N such that, for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives, a computation exists involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ remote memory references to enter and exit its critical section.*[4]

One may wonder whether a $\Omega(\min(k, \log N/ \log\log N))$ lower bound follows from our two lower bounds. Unfortunately, that is not the case, since we have shown that $\Omega(k)$ RMR time complexity is required *provided* N is sufficiently large.

1.4.3 Nonatomic Mutual Exclusion

As explained in Section 1.3, the only prior nonatomic local-spin algorithm is that of Anderson [11], which has $\Theta(N)$ RMR time complexity. Thus, we are led to the following two questions.

- Is it possible to devise a nonatomic local-spin algorithm with $\Theta(\log N)$ time complexity, *i.e.*, that matches the most efficient atomic algorithm known?
- Is it possible to devise an *adaptive* nonatomic algorithm?

Both questions are answered in this dissertation. We answer the first question in the affirmative by presenting a $\Theta(\log N)$ nonatomic algorithm, which is derived from Yang and Anderson's arbitration-tree algorithm by means of simple transformations. On the other hand, the answer to the second question is negative. We show this by proving that any nonatomic algorithm must have a single-process execution in which that process accesses $\Omega(\log N/ \log\log N)$ distinct variables. Therefore, adaptive algorithms are impossible even if caching techniques are used to avoid accessing the interconnection network.

[4]The actual bound on N is given in Theorem 6.1 as $N \geq (2k+4)^{2(2^k-1)}$.

1.4.4 Mutual Exclusion with *fetch-and-ϕ* Primitives

As shown in Section 1.1, constant-time local-spin mutual algorithms are known that use primitives such as *fetch-and-increment* and *fetch-and-store*. The existence of these constant-time algorithms gives rise to a number of intriguing questions regarding mutual exclusion algorithms. Is it possible to devise an $O(1)$ algorithm for DSM machines that uses a single *fetch-and-ϕ* primitive? Can such an algorithm be devised using primitives other than *fetch-and-increment* and *fetch-and-store*? Is it possible to automatically transform a local-spin algorithm for CC machines so that it has the same RMR time complexity on DSM machines? Can we devise a *ranking* of synchronization primitives that indicates the singular characteristic of a primitive that enables a certain RMR time complexity (for mutual exclusion) to be achieved? Such a ranking would provide information relevant to the implementation of *blocking* synchronization mechanisms that is similar to that provided by Herlihy's wait-free hierarchy [41], which is relevant to *nonblocking* mechanisms.[5]

We partially address these questions by presenting a generic N-process *fetch-and-ϕ*-based local-spin mutual exclusion algorithm that has $O(1)$ RMR time complexity on both CC and DSM machines. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$. Informally, a primitive of rank r has sufficient symmetry-breaking power to linearly order up to r invocations of that primitive. This generic algorithm breaks new ground because it shows that $O(1)$ RMR time complexity is possible using a wide range of primitives, on both CC and DSM machines. By applying our generic algorithm within an arbitration tree, one can easily construct a $\Theta(\max(1, \log_r N))$ algorithm using any primitive of rank $r \geq 2$. Furthermore, by combining this arbitration-tree algorithm with our adaptive algorithm presented in Chapter 4, one can easily construct an $O(\min(k, \max(1, \log_r N)))$ adaptive algorithm (where k is contention), using any primitive of rank $r \geq 2$.

[5]In Herlihy's hierarchy, a primitive has *consensus number n* if it can be used to implement wait-free consensus for n, but not $n+1$, processes. In the *consensus* problem, each process starts with its own input value, and every process must eventually agree on the same output value, which must be the input value of some participating process. An object's ranking is determined by its consensus number.

Herlihy's hierarchy is concerned with *computability*: if the consensus number of a primitive (or object) X is higher than that of a primitive (or object) Y, then X can be used to implement Y (in a non-blocking manner) but not *vice versa*. The ranking suggested here is not concerned with computability, but rather time complexity. Nonetheless, both rankings provide information concerning the usefulness of primitives. Herlihy's hierarchy indicates which primitives should be supported in hardware if one is interested in implementing nonblocking algorithms; the proposed ranking indicates which primitives should be supported in hardware if one is interested in implementing scalable mutual exclusion algorithms.

For primitives that meet a certain additional condition, we present a $\Theta(\log N/\log\log N)$ algorithm, which is time-optimal for certain primitives of contant rank. This algorithm can also be made adaptive in a similar way.

1.5 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we describe prior work on shared-memory mutual exclusion. In Chapter 3, we present a simple transformation of Yang and Anderson's algorithm that results in optimal space complexity. In Chapter 4, we present an adaptive mutual exclusion algorithm with atomic reads and writes. In Chapter 5, we present our time-complexity lower bound for generic mutual exclusion algorithms based on reads, writes, and comparison primitives. In Chapter 6, a time-complexity lower bound for *adaptive* mutual exclusion algorithms is given. In Chapter 7, we present a mutual exclusion algorithm based on *nonatomic* reads and writes, together with a very close time-complexity lower bound. In Chapter 8, we describe mutual exclusion algorithms based on generic *fetch-and-*ϕ primitives. Finally, we conclude and discuss future directions for this research in Chapter 9. Detailed proofs of the lower bounds given in Chapters 5 and 7, as well as correctness proofs for the algorithms given in Chapters 4, 7, and 8, are given in appendices.

CHAPTER 2

Related Work*

In this chapter, we survey prior research on shared-memory mutual exclusion. As mentioned in Chapter 1, we will consider only busy-waiting algorithms that do not rely on operating-system services. The first such algorithm was published by Dijkstra in 1965 [34]. Dijkstra's algorithm, which is based on an earlier unpublished two-process algorithm by Dekker, is livelock-free but not starvation-free. A related algorithm published by Knuth in 1966 was the first starvation-free solution [49]. In the years since the publication of Dijksta's and Knuth's algorithms, many other algorithms have been proposed. Many of these algorithms are described in a survey by Michel Raynal in 1986 [72]. A more detailed description of the algorithms mentioned in this chapter, as well as other recent algorithms, can be found in [20].

In recent research on shared-memory mutual exclusion, local-spin algorithms have been one of the major research trends. Most early shared-memory algorithms employ somewhat complicated busy-waiting loops in which many shared variables are read and written. Under contention, such busy-waiting loops generate excessive traffic on the processors-to-memory interconnection network, resulting in poor performance. In local-spin mutual exclusion algorithms, this problem is avoided by requiring all busy-waiting loops to be read-only loops in which only variables cached or stored locally are accessed. We survey work on local-spin algorithms in Section 2.2, after first presenting some needed definitions in Section 2.1.

Another major research trend is work on "fast" mutual exclusion algorithms, namely, algorithms in which a process executes a constant-time "fast path" in the absence of contention. This line of research was motivated by the widely accepted belief that

*The contents of this chapter are adapted from the following survey paper.
[20] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2–3):75–110, September 2003.

"contention for a critical section is rare in well-designed systems" [56]. In Section 2.3, an overview of research on such algorithms is presented.

Over the years, work on fast mutual exclusion algorithms evolved into a broader study of "adaptive" algorithms. In many fast algorithms, there is a sudden jump in time complexity between the contention-free and contention-present cases. In an adaptive algorithm, the rise in time complexity as contention increases is more gradual. Research on such algorithms is surveyed in Section 2.4.

In early work on the mutual exclusion problem, Lamport noted the circularity inherent in algorithms that require accesses of shared memory to be atomic [51]. In the same paper, he presented an algorithm that is correct even if memory accesses are nonatomic. Many nonatomic algorithms have been devised since then. A brief survey of this work is given in Section 2.5.

Reducing time complexity has been an overriding theme in all of the work described above. In work on local-spin algorithms, time complexity is measured by counting memory accesses that cause interconnection network traffic. In work on fast mutual exclusion algorithms, time complexity in the absence of contention is the primary concern. In work on adaptive algorithms, the goal is to minimize time complexity as a function of contention. Given this emphasis on time, it is not surprising that several researchers began to investigate fundamental limits on time complexity through work on lower bounds. Space-complexity bounds has also received related interest. Research on time- and space-complexity lower bounds is discussed in Section 2.6.

2.1 Preliminaries

Before presenting any algorithms, we first define our execution model and describe notational conventions that will be used in the rest of this dissertation. A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program consisting of labeled statements. (We sometimes refer to such statements as *operations*.) Each *variable* of a concurrent program is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may be accessed by more than one process. In the code listings we present, private variables are uncapitalized and shared variables are capitalized. To distinguish private variables of different processes in our algorithm descriptions, we sometimes use the notation $p.v$ to refer to the private variable v of process p. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k.

A program's semantics is defined by its set of "fair histories." The definition of a fair history, which is given below, formalizes the requirement that each statement of a program is subject to weak fairness. Before giving the definition of a fair history, we introduce a number of other concepts; all of these definitions apply to a given concurrent program on atomic systems. (Only atomic statement execution is considered here. Our notion of a nonatomic system is formally defined in Chapter 7.)

A *state* is an assignment of values to the variables of the program. One or more states are designated as *initial states*. If state u can be reached from state t via the execution of statement s, then we say that s is *enabled* at state t and we write $t \xrightarrow{s} u$. If statement s is not enabled at state t, then we say that s is *disabled* at t. A *history* is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state. A history may be either finite or infinite; in the former case, it is required that no statement be enabled at the last state of the history. We say that a history satisfies *weak fairness*, or simply, is *fair*, if it is finite or if it is infinite and each statement is either disabled at infinitely many states of the history or is infinitely often executed in the history [57]. Note that this fairness requirement implies that each continuously-enabled statement is eventually executed.

Each mutual exclusion algorithm we consider is specified as a concurrent program with N processes, each with the following structure.

```
while true do
    Noncritical Section;
    Entry Section;
    Critical Section;
    Exit Section
od
```

The conditions imposed on these code sections were described in Chapter 1. The Exclusion property is required to hold in any history, while the Livelock-freedom and Starvation-freedom properties are required to hold only in fair histories. Unless specified otherwise, we will assume that any solution to the mutual exclusion problem is required to be starvation-free.

Throughout this dissertation, we assess space complexity by counting words of memory (not bits). We use several different time complexity measures, which are introduced as needed.

Some of the algorithms we consider employ read-modify-write synchronization primitives. These primitives execute atomically, *i.e.*, they cause a single state transition. The primitives we consider include *fetch-and-increment*, *fetch-and-decrement*, *fetch-and-add*, *fetch-and-store*, and *compare-and-swap*. These primitives are defined below. In these definitions, *Var* denotes a shared variable, and *val*, *old*, and *new* are used as input and output parameters. In the definition of *compare-and-swap*, *Var*, *old*, and *new* are assumed to be type-consistent. We assume that *Var* is passed by reference. (That is, the *address* of *Var* is actually passed as an argument.)

Primitives *compare-and-swap* and *test-and-set* are ordinarily defined to return a boolean value indicating if the comparison succeeded. In this dissertation, we instead assume that each returns the accessed variable's original value, as in [41]. It is straightforward to modify any algorithm that uses the boolean versions of these primitives to use the versions considered in this dissertation.

```
fetch-and-increment(Var: integer) returns integer
  old := Var;
  Var := Var + 1;
  return(old)

fetch-and-decrement(Var: integer) returns integer
  old := Var;
  Var := Var − 1;
  return(old)

fetch-and-add(Var: integer, val: integer constant)
  Var := Var + val

fetch-and-store(Var, new) returns typeof(Var)
  old := Var;
  Var := new;
  return(old)

test-and-set(bit: boolean) returns boolean
  if bit = false then bit := true; return false
                 else return true
  fi

compare-and-swap(Var, old, new) returns boolean
  if Var = old then Var := new; return old
                else return Var
  fi
```

Throughout this dissertation, the statement "**await** B," where B is a boolean expression, is used as a shorthand for the busy-waiting loop "**while** $\neg B$ **do** /∗ null ∗/ **od**." Also, we use $\log n$ to denote $\log_2 n$ (base-2 logarithm), and use $\log^k n$ to denote $(\log_2 n)^k$.

2.2 Local-spin Algorithms

In local-spin mutual exclusion algorithms, all busy waiting is by means of read-only loops in which one or more "spin variables" are repeatedly tested. Such spin variables must be *locally accessible*, *i.e.*, they can be accessed without causing message traffic on the processors-to-memory interconnection network. As explained in Chapter 1, two architectural paradigms have been considered in the literature that allow shared variables to be locally accessed: distributed shared-memory (DSM) machines and cache-coherent (CC) machines. Both are illustrated in Figure 1.1. As explained in Chapter 1, we define the *RMR* (*remote memory reference*) *time complexity* of a mutual exclusion algorithm to be the worst-case number of remote memory references by one process in order to enter and then exit its critical section.

The first local-spin algorithms were "queue-lock" algorithms in which read-modify-write primitives are used to enqueue blocked processes onto the end of a "spin queue" [23, 38, 61]. In each of these algorithms, a process enqueues itself by using a read-modify-write primitive to update a shared "tail" pointer; a process's predecessor (if any) in the queue is indicated by the primitive's return value. A process in the spin queue waits (if necessary) until released by its predecessor. In Section 2.2.1 below, three queue-lock algorithms are considered in detail and a brief overview of several related algorithms is presented. The three algorithms covered in detail vary in the synchronization primitives used, and whether spinning is local on both CC and DSM systems. In each, a constant number of remote memory references is required per critical-section execution, provided spinning is local.

Yang and Anderson later called into question the necessity of strong synchronization primitives by presenting an algorithm with comparable performance that uses only read and write operations [84]. In terms of RMR time complexity, however, this algorithm is somewhat inferior to the queue locks mentioned above, as it requires $\Theta(\log N)$ remote memory references per critical-section execution. This algorithm and a few other related algorithms are described in Section 2.2.2.

```
const
    $has\_lock = 0$;
    $must\_wait = 1$

shared variable
    $Slots$: **array**$[0..N-1]$ **of** $\{has\_lock,\ must\_wait\}$;
    $Next\_slot$: **integer initially** $0$

initially
    $Slots[0] = has\_lock\ \wedge$
    $(\forall k : 0 < k < N :: Slots[k] = must\_wait)$

process $p$    /* $0 \le p < N$ */

private variable
    $my\_place$: **integer**

    while $true$ do
1:      Noncritical Section;
2:      $my\_place := fetch\text{-}and\text{-}increment(Next\_slot)$;
3:      if $my\_place = N-1$ then
4:          $fetch\text{-}and\text{-}add(Next\_slot,\ -N)$
        fi;
5:      $my\_place := my\_place$ mod $N$;
6:      await $Slot[my\_place] = has\_lock$;    /* spin */
7:      $Slots[my\_place] := must\_wait$;
8:      Critical Section;
9:      $Slot[my\_place + 1$ mod $N] := has\_lock$
    od
```

Figure 2.1: ALGORITHM TA: Array-based queue lock using *fetch-and-increment* and *fetch-and-add*.

2.2.1 Algorithms that use *fetch-and-ϕ* Primitives

We begin by considering two queue-lock algorithms, by T. Anderson [23] and by Graunke and Thakkar [38], in which the spin queue is stored in a shared array. Each of these algorithms has $O(1)$ RMR time complexity under the CC model, but unbounded RMR time complexity under the DSM model. In the third algorithm we consider, the spin queue is stored as a shared linked list. This algorithm, which was proposed by Mellor-Crummey and Scott [61], has $O(1)$ RMR time complexity under both the CC and DSM models.

ALGORITHM TA. T. Anderson's algorithm [23], denoted ALGORITHM TA,[1] is shown in Figure 2.1. ALGORITHM TA uses both *fetch-and-increment* and *fetch-and-add*. (A

[1]Throughout this dissertation, an algorithm is named after the initial of its author(s) (if it is published by others) or its chief characteristic (if it belongs to the work presented in this dissertation).

fetch-and-increment primitive that takes the value to add as input can be used in place of *fetch-and-add*; in this case, the return value of *fetch-and-increment* is ignored.) The spin queue is defined by an array of "slots." These slots are indexed from 0 to $N-1$. The next free slot at the tail of the queue is indicated by the shared variable *Next_slot*. A process p enqueues itself onto the end of the spin queue by simply using *fetch-and-increment* to increment *Next_slot* (statement 2). In addition to updating *Next_slot*, the *fetch-and-increment* operation returns the slot for p to use, which is stored in the private variable $p.my\_place$. The main complication to be dealt with occurs when some process increments *Next_slot* beyond slot $N-1$. In this case, the process q that increments *Next_slot* from $N-1$ to N will find $q.my\_place = N-1$ at statement 3, and then execute the *fetch-and-add* operation at statement 4 to "correct" the value of *Next_slot*. (Note that *Next_slot* may be incremented at most $N-1$ times by other processes before this correcting step is performed. This is because any such process is enqueued after q and thus is blocked until q finishes its critical section.) Statement 5 ensures that the value of $q.my\_place$ ranges over $\{0, \ldots, N-1\}$.

The value of each slot ranges over $\{has\_lock,\ must\_wait\}$. A process in its entry section waits until its slot has the value *has_lock* (statement 6). If there is a successor to process p in the spin queue, then its slot is $p.my\_place + 1 \ \mathbf{mod}\ N$. If a successor does exist, then it is granted the lock when p executes statement 9. If no successor exists, then statement 9 ensures that the lock will be granted to the next process that performs the *fetch-and-increment* operation at statement 2. (The initial conditions also ensure this.) Statement 7 is executed by p to reinitialize its slot for future use.

The RMR time complexity of Algorithm TA is clearly determined by the number of remote memory references generated by statement 6. Under the CC model, statement 6 generates a constant number of remote memory references. To see this, note that the first read of $Slots[p.my\_place]$ creates a cached copy. If $Slots[p.my\_place] = must\_wait$ holds, then $Slots[p.my\_place]$ will remain cached until it is either invalidated or updated by another process, but this occurs only when p's predecessor in the spin queue executes statement 9, which establishes $Slots[p.my\_place] = has\_lock$. At this point, an additional read of $Slots[p.my\_place]$ causes p's waiting to terminate. Under the DSM model, Algorithm TA has unbounded RMR time complexity. This is because different processes spin on different memory locations at different times, and hence, these locations cannot be statically allocated so that all spins are local. Thus, we have the following theorem.

Theorem 2.1 (T. Anderson) *The mutual exclusion problem can be solved with $O(1)$ RMR time complexity using fetch-and-increment under the CC model.* □

Algorithm GT. Graunke and Thakkar's algorithm [38], denoted Algorithm GT, is also an array-based queue lock, with a structure similar to the previous algorithm. In this case, however, the enqueue operation is implemented using *fetch-and-store*. Recall that in Algorithm TA, the association of slots to processes is not fixed but the ordering of the slots comprising the queue of waiting processes is (*e.g.*, if slot 0 is not at the end of the queue, then slot 1 is the slot following it). Here, the association of slots to processes *is* fixed but the ordering of the slots comprising the queue varies dynamically. In particular, each slot is defined by a boolean value and is "owned" by a unique process. A process enqueues itself by appending its slot to the end of the queue. A waiting process uses its *predecessor*'s slot as a spin variable, *i.e.*, a process p with a predecessor q in the queue waits until q updates the slot owned by q. This is different from Algorithm TA, where each process uses the slot it obtains from the *fetch-and-increment* operation as its spin variable.

The RMR time complexity of Algorithm GT is $O(1)$ under the CC model. Under the DSM model, Algorithm GT has unbounded RMR time complexity. This is because each process waits on information stored within the slot of its predecessor in the spin queue. This information cannot be statically allocated so that all spins are local. Thus, from Algorithm GT, we have the following theorem.

Theorem 2.2 (Graunke and Thakkar) *The mutual exclusion problem can be solved with $O(1)$ RMR time complexity using fetch-and-store under the CC model.* □

Algorithm MCS. The final queue-lock algorithm we consider in detail is a linked-list-based algorithm due to Mellor-Crummey and Scott [61]. This algorithm, denoted Algorithm MCS, is shown in Figure 2.2. (In this figure, a->b is used as a shorthand for $(*a).b$, where a is a pointer to a record with component b.) Algorithm MCS employs both *fetch-and-store* and *compare-and-swap*.

Each entry in the linked list is called a *Qnode*, and each process has its own dedicated *Qnode* (which is assumed to be stored locally, if the algorithm is implemented on a DSM machine). The *Qnode* for each process p has two components: a pointer to p's successor in the spin queue (if any), and a boolean variable *locked*, which is p's spin location. The shared variable *Tail* points to the last *Qnode* in the queue. *Tail* is initially *NIL*.

```
type
  Qnode = record next: pointer to Qnode;  locked: boolean end        /* stored in one word */

shared variable
  Nodes: array[0..$N-1$] of Qnode;                        /* Nodes[p] is stored locally to process p */
  Tail: pointer to Qnode initially NIL

process p    /* $0 \leq p < N$ */

private constant
  my_node = &Nodes[p]

private variable
  pred: pointer to Qnode

  while true do
1:  Noncritical Section;
2:  my_node -> next := NIL;
3:  pred := fetch-and-store(Tail,  my_node);
4:  if pred $\neq$ NIL then
5:    my_node -> locked := true;
6:    pred -> next := my_node;
7:    await $\neg$my_node -> locked                      /* spin until granted the lock by predecessor */
    fi;
8:  Critical Section;
9:  if my_node -> next = NIL then
10:   if compare-and-swap(Tail, my_node, NIL) $\neq$ my_node then
11:     await my_node -> next $\neq$ NIL;                          /* spin until next field is updated */
12:     my_node -> next -> locked := false
      fi
    else
13:   my_node -> next -> locked := false
    fi
  od
```

Figure 2.2: Algorithm MCS: List-based queue lock using *fetch-and-store* and *compare-and-swap*.

A process p threads itself onto the end of the spin queue by performing the *fetch-and-store* operation at statement 3. This *fetch-and-store* causes *Tail* to point to p's *Qnode* and also returns to p the previous value of *Tail*. If p threads itself onto a nonempty spin queue, then this previous value gives p's predecessor in the queue. In this case, p initializes its spin location (statement 5), updates the *next* pointer of its predecessor (statement 6), and then busy-waits until released by its predecessor (statement 7).

In its exit section, p must release its successor in the spin queue, if subsequent processes in the queue do indeed exist. If $p.my\_node$ `->` $next \neq NIL$ holds when p executes statement 9, then p can easily update its successor's spin location (statement 13). However, if $p.my\_node$ `->` $next = NIL$ holds, then a potential problem arises. In particular, it may be the case that p has no successor, or it may be the case that it does have a successor, but that process has not yet updated p's *next* field. This ambiguity is resolved by the *compare-and-swap* on *Tail* performed at statement 10. If p indeed has no successor, then *Tail* must still point to p's *Qnode*, in which case the *compare-and-swap* succeeds. On the other hand, if p has a successor, then the *compare-and-swap* fails, and p executes statements 11–12. Statement 11 causes p to wait until its *next* pointer has been updated by its successor. (This is one of the few mutual exclusion algorithms found in the literature in which a process may wait in its exit section.) Statement 12 then updates its successor's spin location.

Because each process has its own dedicated spin location, it should be clear that the algorithm has $O(1)$ RMR time complexity under either the CC or DSM model. Thus, we have the following.

Theorem 2.3 (Mellor-Crummey and Scott) *The mutual exclusion problem can be solved with $O(1)$ RMR time complexity using fetch-and-store and compare-and-swap under either the CC or DSM model.* □

One problem with Algorithm MCS is that it relies on two synchronization primitives, which may limit its applicability. To circumvent this problem, Mellor-Crummey and Scott also presented a variant that uses only *fetch-and-store* [61]. Unfortunately, this variant is only livelock-free. However, the authors argue that starvation should be very unlikely in practice.

Other related algorithms. A number of researchers have proposed extensions to the three algorithms covered so far that support process priorities or that tolerate process preemptions [13, 32, 44, 50, 60, 73, 82, 83]. Priorities can be supported either by requiring the spin queue to be priority ordered, or by requiring each process in its exit section to completely scan the queue to find the highest-priority waiting process. In the former case, a process must have the ability to scan the queue and insert its queue record at the position indicated by its priority. Priority-based systems are often multiprogrammed, *i.e.*, there may be multiple processes bound to the same processor. In multiprogrammed systems, preemptions may be common. Preemptions are especially problematic for queue locks, because a preempted process may delay every process after it in the spin queue. Most proposals for dealing with preemptions rely on the kernel to deactivate or remove the queue record of a preempted process.

A restricted form of priority that has been well-studied occurs in algorithms for reader/writer synchronization [31]. Reader/writer synchronization is a generalization of mutual exclusion in which each process is classified as either a reader or a writer. Readers may execute their critical sections simultaneously, but writers require exclusive access. Because readers and writers have different requirements, it is necessary to give readers priority over writers or *vice versa*. After the development of Algorithm MCS, Mellor-Crummey and Scott presented several extensions of that algorithm that support reader/writer synchronization [62].

Other authors have investigated algorithms that use circular waiting lists [36, 42, 43]. Note that, in Algorithm MCS, each process p finds its predecessor in the list by performing the *fetch-and-store* at statement 3. However, p cannot identify its successor (if any) until that successor first updates p's *next* field. By using a circular list, this problem can be eliminated, because all nodes can be reached by traversing through the list by reading predecessor pointers.

Fu and Tzeng [36, 43] considered a circular-list algorithm where processes are arranged in a tree, each node of which contains a circular waiting list. Each process in its entry section enqueues itself onto a leaf circular list. A process that is at the head of the circular list at some node ascends the tree, merging the circular list of that node with the circular list of its parent node. It is argued that the tree structure eliminates hot-spot contention, leading to better performance. (*Hot-spot contention* occurs when many processes repeatedly access the same shared variable, or variables stored in the same memory module. Hot-spot contention can lead to degraded performance for all memory accesses, even those that do not target the hot spot [71].) In other work, Huang [42] presented a circular list algorithm that uses only *fetch-and-store* and that has constant *amortized* RMR time complexity on both CC and DSM systems (*i.e.*, the number of remote memory references in a history divided by the number of critical-section entries in that history is constant).

In other related work, extensions of some of the queue-based algorithms discussed above were recently proposed in which timeout mechanisms are incorporated [75, 76]. Such mechanisms can be used by a process to abandon its lock request if it has waited too long (*e.g.*, if a deadline has passed).

As noted earlier, Algorithms TA and GT have $O(1)$ RMR time complexity only under the CC model. In later work, Craig [32] and Landin and Hagersten [59] independently proposed a queue-based algorithm based on *fetch-and-store*. While Landin and

Hagersten considered only CC machines, Craig presented constant-time variants of the algorithm for both CC and DSM machines. Thus, Theorem 2.2 can be strengthened to also apply to DSM systems. In Chapter 8, we show that constant-time algorithms can be constructed for DSM systems using any of a large class of primitives that includes *fetch-and-increment*. Thus, Theorem 2.1 also can be strengthened to apply to DSM systems.

2.2.2 Algorithms that use Only Reads and Writes

Algorithms TA, GT, and MCS were the first local-spin mutual exclusion algorithms to be published, and each requires one or more read-modify-write primitives. This led some researchers to question whether such primitives were in fact *necessary* for local-spin synchronization. In 1993, Anderson showed that this was not the case by presenting an $\Theta(N)$ algorithm that uses only read and write operations [11]. Although this algorithm showed that local-spin synchronization without strong primitives was possible in principle, its RMR time complexity is significantly higher than Algorithms TA, GT, and MCS. In subsequent work, Yang and Anderson narrowed this time-complexity gap by presenting an $\Theta(\log N)$ algorithm based on reads and writes [84]. From the lower bound of Chapter 5, it follows that the RMR time complexity of Yang and Anderson's algorithm is within a factor of $\Theta(\log \log N)$ of optimality for algorithms that use only atomic reads and writes.

In the rest of this section, an overview is given of Yang and Anderson's algorithm, hereafter denoted Algorithm YA [84]. The earlier algorithm of Anderson [11] is considered in Section 2.5.

Algorithm YA. In [45], Kessels proposed implementing N-process mutual exclusion by using instances of a two-process algorithm (which, in Kessels' algorithm, is not a local-spin algorithm) in a binary arbitration tree. Associated with each link in the tree is an entry section and an exit section. The entry and exit sections associated with the two links connecting a given node to its children constitute a two-process mutual exclusion algorithm. Initially, all processes are "located" at the leaves of the tree. To enter its critical section, a process is required to traverse a path from its leaf up to the root, executing the entry section of each link on this path. Upon exiting its critical section, a process traverses this path in reverse, this time executing the exit section of each link.

```
shared variables
  $C$: array$[u, v]$ of $\{u, v, \bot\}$ initially $\bot$;
  $P$: array$[u, v]$ of $0..2$ initially $0$;
  $T$: $\{u, v\}$
```

```
process $u$

while $true$ do
   1: Noncritical Section;
   2: $C[u] := u$;
   3: $T := u$;
   4: $P[u] := 0$;
   5: if $C[v] \neq \bot$ then
   6:    if $T = u$ then
   7:       if $P[v] = 0$ then
   8:          $P[v] := 1$;
            fi;
   9:       await $P[u] \geq 1$; /* spin */
  10:       if $T = u$ then
  11:         await $P[u] = 2$ /* spin */
            fi
         fi
      fi;
  12: Critical Section;
  13: $C[u] := \bot$;
  14: if $T \neq u$ then
  15:    $P[v] := 2$
      fi
od
```

```
process $v$

while $true$ do
   1: Noncritical Section;
   2: $C[v] := v$;
   3: $T := v$;
   4: $P[v] := 0$;
   5: if $C[u] \neq \bot$ then
   6:    if $T = v$ then
   7:       if $P[u] = 0$ then
   8:          $P[u] := 1$
            fi;
   9:       await $P[v] \geq 1$; /* spin */
  10:       if $T = v$ then
  11:          await $P[v] = 2$ /* spin */
            fi
         fi
      fi;
  12: Critical Section;
  13: $C[v] := \bot$;
  14: if $T \neq v$ then
  15:    $P[u] := 2$
      fi
od
```

Figure 2.3: ALGORITHM YA-2: A two-process version of ALGORITHM YA.

ALGORITHM YA is based on the arbitration-tree approach of Kessels. For this approach to work in a DSM system when all busy-waiting is by local spinning, the two-process algorithm being used must provide a mechanism that allows a process to deduce the process (if any) with which it must compete. (If a process incorrectly determines its competitor at some node of the tree, then it may end up writing to a spin variable of a process that has not even accessed that node!) The two-process version of ALGORITHM YA provides such a mechanism. A slightly-simplified version of the two-process algorithm, denoted ALGORITHM YA-2, is shown in Figure 2.3. In this figure, the two processes are denoted by the identifiers u and v, which are assumed to be distinct, nonnegative integer values.

The two-process algorithm employs five shared variables, $C[u]$, $C[v]$, T, $P[u]$, and $P[v]$. Variable $C[u]$ ranges over $\{u, v, \bot\}$ and is used by process u to inform process v of its intent to enter its critical section. Observe that $C[u] = u \neq \bot$ holds while process u is at statements 3–13, and $C[u] = \perp$ holds otherwise. Variable $C[v]$ is used similarly. Variable T ranges over $\{u, v\}$ and is used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. The algorithm ensures that the two processes enter their critical sections according to the order in which they update T. Variable $P[u]$ ranges over $\{0, 1, 2\}$ and is used by process u whenever it needs to busy-wait. Variable $P[v]$ is used similarly by process v. (Note that these are statically-allocated spin variables; hence, on DSM machines, they can be stored locally.)

When process u wants to enter its critical section, it informs process v of its intention by establishing $C[u] = u$. Then, process u assigns its identifier u to the tie-breaker variable T, and initializes its spin location $P[u]$. If process v has not shown interest in entering its critical section, *i.e.*, if $C[v] = \perp$ holds when u executes statement 5, then process u proceeds directly to its critical section. Otherwise, u reads the tie-breaker variable T. If $T \neq u$, which implies that $T = v$, then u can enter its critical section, as the algorithm prohibits v from entering its critical section when $C[u] = u \wedge T = v$ holds (recall that ties are broken in favor of the first process to update T). If $T = u$ holds, then either process v executed statement 3 before process u, or process v has executed statement 2 but not statement 3. In the first case, u should wait until v exits its critical section, whereas, in the second case, u should be able to proceed to its critical section. This ambiguity is resolved by having process u execute statements 7–11. Statements 7–8 are executed by process u to release process v in the event that it is waiting for u to update the tie-breaker variable (*i.e.*, v is busy-waiting at statement 9). Statements 9–11 are executed by u to determine which process updated the tie-breaker variable first. Note that $P[u] \geq 1$ implies that v has already updated the tie-breaker, and $P[u] = 2$ implies that v has finished its critical section. To handle these two cases, process u first waits until $P[u] \geq 1$ (*i.e.*, until v has updated the tie-breaker), re-examines T to see which process updated T last, and finally, if necessary, waits until $P[u] = 2$ (*i.e.*, until process v finishes its critical section).

After executing its critical section, process u informs process v that it is finished by establishing $C[u] = \perp$. If $T = v$, in which case process v is trying to enter its critical section, then process u updates $P[v]$ so that v does not wait.

It is straightforward to construct a general N-process algorithm, by embedding instances of this two-process algorithm within a binary arbitration tree. (A general

N-process algorithm, denoted Algorithm YA-N, can be found in Figure 3.2.) Thus, we have the following theorem.

Theorem 2.4 (Yang and Anderson) *The mutual exclusion problem can be solved with* $\Theta(\log N)$ *RMR time complexity using only reads and writes under either the CC or DSM model.* □

Other related algorithms. As mentioned before, Anderson also devised a $\Theta(N)$ algorithm that uses only reads and writes [11], which was also the first nonatomic algorithm in which all spins are local; this algorithm is discussed in Section 2.5.

Aside from the algorithms already mentioned, the only other published local-spin mutual exclusion algorithm that uses only reads and writes that we know of is one by Tsay [80]. Tsay's algorithm is very similar to Algorithm YA and also the earlier algorithm of Anderson [11]. Tsay derives his algorithm through a series of transformations from well-known algorithm of Peterson [69].

Zhang, Yan, and Castañeda have conducted an extensive evaluation of several (read/write)-based mutual exclusion algorithms [85]. Algorithm YA is one of the algorithms tested by them. Their evaluation is based on several metrics that take into account the effects of architectures, systems, and software implementations. In addition, they present three new algorithms, including two tree-based algorithms, that incorporate both local-spin and non-local-spin techniques.

2.3 Fast Mutual Exclusion

In 1987, Lamport devised a novel mutual exclusion algorithm that requires only seven memory accesses in the absence of contention [56]. Algorithms such as this, in which a process executes a constant-time "fast path" in the absence of contention, are known as "fast" mutual exclusion algorithms. (When determining contention-free time complexities, all memory references are counted, local and remote.) Each of the *fetch-and-*ϕ-based algorithms covered in Section 2.2.1 clearly has constant time complexity in the absence of contention. Thus, time complexity in the absence of contention is a non-issue if suitable *fetch-and-*ϕ primitives are available. For this reason, the term "fast mutual exclusion algorithm" is usually applied only to algorithms that use only reads and writes. In this section, we present an overview of research to date on such algorithms.

2.3.1 Algorithm L: Lamport's Fast Mutual Exclusion Algorithm

Lamport's fast mutual exclusion algorithm, hereafter denoted Algorithm L, is shown in Figure 2.4(a). In this algorithm, a fast-path process reaches its critical section by executing (only) statements 2, 3, 4, 8, and 9. Of these, statements 3, 4, 8, and 9 are of special significance. These statements are shown separately in Figure 2.4(c), where they are used to define a "black box" element called a *splitter*, which is illustrated in Figure 2.4(b). (The term "splitter" is not due to Lamport; it was first abstracted as a "black box" by Moir and Anderson [66], and first called a "splitter" by Attiya and Fouren [25].) In the following paragraphs, we consider some properties of the splitter that make it so useful, and then show how these properties ensure the correctness of Algorithm L.

The splitter element. Each process that invokes the splitter code either stops, moves down, or moves right. (The move is defined by the value assigned to the private variable *dir*.) One of the key properties of the splitter that makes it so useful is the following: if several processes invoke a splitter, then at most one of them can stop at that splitter. To see why this property holds, suppose to the contrary that two processes p and q stop. Let p be the process that executed statement 4 last. Because p found that $X = p$ held at statement 4, X is not written by any process between p's execution of statement 1 and p's execution of statement 4. Thus, q executed statement 4 before p executed statement 1. This implies that q executed statement 3 before p executed statement 2. Thus, p must have read $Y = \mathit{false}$ at statement 2 and then assigned "$p.dir := \mathit{right}$," which is a contradiction. Similar arguments can be applied to show that if n processes invoke a splitter, then at most $n-1$ can move right, and at most $n-1$ can move down.

Because of these properties, the splitter element and related mechanisms have proven to be immensely useful in wait-free algorithms for renaming [2, 4, 26, 25, 27, 66, 67]. Renaming algorithms are used to "shrink" the name space from which process identifiers are taken. Such algorithms can be used to speed up concurrent computations with loops that iterate over process identifiers. Because of the splitter's properties, it is possible to solve the renaming problem by interconnecting a collection of splitters in a grid [67].

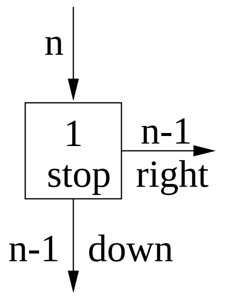
```
shared variable
    $B$: array[1..$N$] of boolean initially $false$;
    $X$: 1..$N$;
    $Y$: 0..$N$ initially 0

process $p$                      /* $1 \leq p \leq N$ */

private variable
    $j$: 1..$N$

while $true$ do
1:  Noncritical Section;
2:  $B[p] := true$;
3:  $X := p$;
4:  if $Y \neq 0$ then
5:      $B[p] := false$;
6:      await $Y = 0$;            /* busy wait */
7:      goto 1
    fi;
8:  $Y := p$;
9:  if $X \neq p$ then
10:     $B[p] := false$;
11:     for $j := 1$ to $N$ do
12:         await $\neg B[j]$     /* busy wait */
        od;
13:     if $Y \neq p$ then
14:         await $Y = 0$;        /* busy wait */
15:         goto 1
        fi
    fi;
16: Critical Section;
17: $Y := 0$;
18: $B[p] := false$
od
```

(a)



(b)

```
/* $X$ and $Y$ are as in part (a) */

process $p$

private variable
    $dir$: {$stop$, $right$, $down$}

1:  $X := p$;
2:  if $Y \neq 0$ then $dir := right$
    else
3:      $Y := p$;
4:      if $X \neq p$ then $dir := down$
                      else $dir := stop$
        fi
    fi
```

(c)

Figure 2.4: **(a)** ALGORITHM L: Lamport's fast mutual exclusion algorithm. **(b)** The splitter element and **(c)** its implementation.

Correctness of ALGORITHM L**.** The splitter's properties also ensure that ALGORITHM L is correct. In particular, because at most one process can stop at a splitter, at most one process at a time can "take the fast path" by reading $Y = 0$ at statement 4 and $X = p$ at statement 9. (This corresponds to the assignment of $dir := stop$ at Figure 2.4(c).) Moreover, if no process takes the fast path during a period of contention, then some process must reach statements 10–15. It can be shown that, of these processes, the last to update the variable Y eventually enters its critical section and then reopens the fast path by assigning $Y := 0$ at statement 17. Thus, the algorithm is livelock-free, giving us the following theorem.

Theorem 2.5 (Lamport) *The mutual exclusion problem can be solved by a livelock-free algorithm that requires only seven memory references in the absence of contention.* □

On the other hand, starvation-freedom is not satisfied, because an unfortunate process may repeatedly find either $Y \neq 0$ at statement 4 or $Y \neq p$ at statement 13, and hence wait forever.

Variations. A number of authors have proposed variants of Algorithm L. Alur and Taubenfeld have shown that the number of memory references in the absence of contention can be reduced to five, if each process has the ability to delay itself by an amount of time that depends on the speeds of other processes [8]. (Fischer initiated the study of such delay-based algorithms. Fischer's algorithm can be found in [56]. In this algorithm, a process repeatedly writes to a common variable X, delays itself, and examines X again. The delay is assumed long enough to ensure that any other competing process must have written to X. Thus, a process reads the value it has written only if there exists no other competing process.) Michael and Scott showed that only two reads and four writes are required in the absence of contention, if processes have the ability to read and write at both full- and half-word granularities [65].

In contrast, Merritt and Taubenfeld proposed modifications that speed up the algorithm in the *presence* of contention, as compared with Algorithm L [63]. In Algorithm L, each process that reaches statement 11 must check the status of every other process. However, in an actual system, the number of processes that actually invoke the algorithm concurrently is likely to be much less than the total number of processes in the system. Merritt and Taubenfeld showed that by using a linked list instead of a simple array scan, the time complexity of this check can be made proportional to the number of contending processes.

2.3.2 Fast Mutual Exclusion with Local Spinning

All of the fast mutual exclusion algorithms discussed above employ busy-waiting loops in which shared variables are both read and written. Thus, while these algorithms are fast in the *absence* of contention, each has unbounded RMR time complexity *under* contention. In this section, we consider fast mutual exclusion algorithms in which all busy-waiting is by local spinning.

The first such algorithm to be published was actually a variant of ALGORITHM YA-N considered earlier in Section 2.2.2 [84]. Unfortunately, in this fast-path variant of ALGORITHM YA-N, RMR time complexity under contention is $\Theta(N)$ instead of $\Theta(\log N)$. In later work [16], Anderson and Kim presented an improved fast-path mechanism that results in $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ RMR time complexity under contention, when used in conjunction with ALGORITHM YA-N. This result is subsumed by an adaptive algorithm devised later, which is presented in Chapter 4.

2.4 Adaptive Algorithms

Although fast mutual exclusion algorithms perform well in the absence of contention, most such algorithms exhibit a sudden rise in time complexity when contention is present. In this section, we consider adaptive algorithms, which are designed to alleviate this problem. Formally, a mutual exclusion algorithm is *adaptive* if its time complexity (under some measure) is a function of the number of contending processes. As is the case with fast algorithms, adaptivity is a non-issue if appropriate synchronization primitives are available. Thus, the term "adaptive" is usually applied only to algorithms that use only reads and writes.

Two notions of contention have been considered in the literature: "interval contention" and "point contention" [2]. These two notions are defined with respect to a history H. The *interval contention* over H is the number of processes that are active in H, *i.e.*, that execute outside of their noncritical sections in H. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. In this section, unless stated otherwise, we let k (k') denote the point (interval) contention experienced by an arbitrary process over a history that starts when that process becomes active and ends when it once again becomes inactive. We also let M denote an upper bound on the maximum number of processes concurrently active in the system (possibly less than N). The algorithms considered in this section are summarized in Table 2.1.

Several different time complexity measures have been applied in work on adaptive algorithms. In defining a meaningful time complexity measure for concurrent algorithms, dealing with potentially unbounded busy-waiting loops is the main difficulty

Algorithm	**System response time**	**Step time complexity**	**RMR/DSM time complexity**	**Space complexity**
Styer [78]	$O(\min(N, k' \log N))$	$O(\min(N, k' \log N))$	∞	$\Theta(N)$
Choy & Singh [30]	$O(k')$	$O(N)$	∞	$\Theta(N)$
Attiya & Bortnikov [24]	$O(\log k)$	$O(k)$	∞	$\Theta(N \log M)$
Attiya & Bortnikov [24]	$O(\log k')$	$O(k')$	∞	$\Theta(M \log M)$
Afek, *et al.* [5]	$O(k^4)$	$O(k^4)$	∞	$\Theta(N^3 + M^3 N)$
Afek, *et al.* [6]	$O((k')^2)$	$O(\min((k')^2, k' \log N))$	$O(\min((k')^2, k' \log N))$	$\Theta(N^2)$
Chapter 4	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$\Theta(N)$

Table 2.1: Comparison of known adaptive algorithms. In this table, k denotes point contention, k' denotes interval contention, and M denotes an upper bound on the maximum number of processes concurrently active in the system (possibly less than N). (Although [5] uses a bounded number of variables, some of these variables are unbounded.) Each algorithm has bounded RMR time complexity on CC machines with write-update caches. Since these algorithms are quite complicated, it is unclear whether they are adaptive on CC machines in general.

to be faced. Indeed, under the standard sequential-algorithms measure of counting all operations, such a busy-waiting loop has unbounded time complexity. This is inevitable under contention [8], and thus the standard sequential measure provides information that is not very interesting or useful.

The *step time complexity* (also called the "remote step complexity") of an algorithm is the maximum number of shared-memory operations required by a process to enter and then exit its critical section, assuming that each "**await**" statement is counted as one operation [78]. This measure simply ignores repeated memory references generated by a process while it waits. The *system response time* is the length of time between the end of a critical section and the beginning of the next critical section, assuming every active process performs at least one step within some constant time bound [30]. By forcing active processes to take steps, unbounded waiting times are precluded, provided each waiting condition in an algorithm is eventually established by some process within a finite number of its own steps. The *amortized system response time* of an algorithm, a measure also proposed in [30], is defined as the average system response time, provided that all k contending processes start execution simultaneously. The RMR time complexity measure is also of interest in work on adaptive algorithms.

Before continuing, let us examine the relationship between RMR time complexity and step time complexity. Under the DSM model, if all **await** statements within an algorithm access only locally-accessible variables, then each statement has $O(1)$ step time complexity and $O(1)$ RMR time complexity.[2] On the other hand, if an **await** statement accesses a non-local variable, then the algorithm's RMR time complexity is obviously unbounded. Under the CC model, any variable accessed in an **await** statement will be brought into a local cache line. Therefore, if write-update cache is used, then an algorithm's RMR time complexity is at most its step time complexity. However, in systems with write-invalidate caches, a single **await** statement might generate a large number of cache misses if the variables it accesses keep changing value without satisfying the **await** condition. The step time complexity measure ignores the remote references caused by these misses. Therefore, in general, an algorithm's step time complexity and RMR time complexity need not be the same.

As mentioned at the end of Section 2.3.1, Merritt and Taubenfeld proposed a variant of Algorithm L in which a linked list of active processes is scanned, rather than an array of per-process status variables [63]. However, their algorithm requires an external mechanism for inserting active processes into the list and for removing ones that are no longer active. (These operations must be done in a critical section in order to ensure correctness. Hence, unless we assume an external mechanism, another solution of the mutual exclusion problem is required.) Thus, it is not a true adaptive algorithm in the sense considered here.

One of the first true adaptive algorithms was an algorithm of Styer that has $O(\min(N, k' \log N))$ step time complexity and $O(\min(N, k' \log N))$ system response time [78]. This algorithm is rather complicated and is not discussed in detail here.

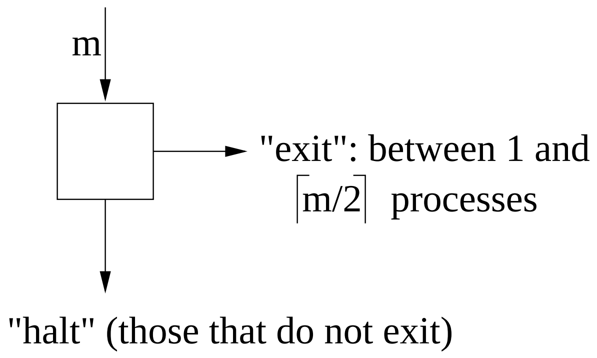
2.4.1 Adaptive Algorithms using Filters

Choy and Singh [30] devised a novel code fragment called a *filter*, which is shown in Figure 2.5. A process executing a filter either exits (succeeds) or halts (fails). A filter satisfies the following two properties.

- **Safety:** If m processes enter the filter, then at most $\lceil m/2 \rceil$ processes exit.
- **Progress:** If some processes enter the filter, then at least one of them exits.

To see that the Progress property holds, let p be the last process to execute statement 1. If no other process exits, then all the halting processes will assign $b :=$ *false*

[2]This conclusion is based upon the assumption that, under the step time complexity measure, each **await** statement has constant cost regardless of the number of variables it accesses. It is not clear if this is reasonable if the number of variables accessed is a function of N. In all papers where step time complexity is used, only **await** statements that access a constant number of variables are considered.


m
"exit": between 1 and ⌈m/2⌉ processes
"halt" (those that do not exit)

(a)

```
shared variable
    turn: 1..N;
    b: boolean initially false

process p         /* $1 \le p \le N$ */
1:  turn := p;
2:  await $\neg b$;
3:  b := true;
4:  if turn $\neq$ p then
5:      b := false;
6:      halt
    else
7:      exit
    fi
```

(b)

Figure 2.5: **(a)** The filter element and **(b)** its implementation.

at statement 5. Therefore, p will eventually find $b = \mathit{false}$ at statement 2 and exit successfully. Now, consider the safety property. Assume that e processes exit. Note that, for any two exiting processes, their executions of statements 1–4 do not overlap. (If an exiting process p executes statement 1 before another exiting process q does, then p must execute statement 4 before q executes statement 1, in order to exit.) While such a process executes statements 1–4, the value of b changes from *false* to *true* at least once. Therefore, the value of b must change from *true* to *false* at least $e - 1$ times, so there are at least $e - 1$ halting processes. The safety property follows from the inequality $e + (e - 1) \le m$.

Leader election using filters. A leader-election algorithm can be constructed by concatenating filters so that only those processes that exit from a filter move to the next. The algorithm is shown in Figure 2.6(a).

We concatenate $\lceil \log_2 N \rceil + 1$ filters, indexed from 0. Since each filter halves the number of exiting processes, at most one process may exit from filter $A[\lceil \log_2 N \rceil]$. When a process p exits from filter $A[i+1]$, it checks filter $A[i]$ in order to determine if p is the only process that has exited from filter $A[i]$. If this is the case, then p elects itself as the leader. (This is why we need one additional filter after $A[\lceil \log_2 N \rceil]$.) The detailed mechanism is explained next.

Each filter has an additional variable c, which is initially *false* and set to *true* when a process fails at that filter. Once c is set to *true*, it remains *true*. Therefore, if process

```
shared variable
    $A$: array$[0..\lceil\log_2 N\rceil + 1]$ of record
        $turn$: $1..N$;
        $b$, $c$: boolean initially $false$ end

process $p$            /* $1 \le p \le N$ */

private variable
    $curr$: $0..\lceil\log_2 N\rceil + 1$

$curr := 0$;
while $true$ do
1:  $A[curr].turn := p$;
2:  await $\neg A[curr].b$;
3:  $A[curr].b := true$;
4:  if $A[curr].turn \neq p$ then
5:      $A[curr].c := true$;
6:      $A[curr].b := false$;
7:      halt
8:  elseif $curr > 0 \land A[curr-1].c$ then
9:      elected
    else
10:     $curr := curr + 1$
    fi
od
```

(a)

```
shared variable
    $A$: array$[0..\infty]$ of record
        $turn$: $1..N$;
        $b$, $c$: boolean initially $false$ end;
    $Entry$: $0..\infty$ initially $0$

process $p$            /* $1 \le p \le N$ */

private variable
    $curr$, $lentry$: $0..\infty$;
    $elected$: boolean

while $true$ do
1:  Noncritical Section;
2:  $elected := false$;
3:  while $\neg elected$ do
4:      $lentry,\ curr := Entry,\ Entry$;
5:      while $\neg elected \land Entry = lentry$ do
6:          $A[curr].turn := p$;
7:          await $(\neg A[curr].b \lor$
8:                  $Entry \neq lentry)$;
9:          if $Entry = lentry$ then
10:             $A[curr].b := true$;
11:             if $A[curr].turn \neq p$ then
12:                 $A[curr].c := true$;
13:                 $A[curr].b := false$;
14:                 await $Entry \neq lentry$
15:             elseif $curr > lentry \land$
                    $A[curr-1].c$ then
16:                 $elected := true$
                else
17:                 $curr := curr + 1$
            fi fi
        od
    od;
18: Critical Section;
19: $Entry := curr + 1$
od
```

(b)

Figure 2.6: **(a)** A leader-election algorithm using filters. **(b)** Unbounded version of Choy and Singh's algorithm [30].

p exits from filter $A[i+1]$ and finds $A[i].c = false$, then no process has (yet) failed at filter $A[i]$. It follows that $A[i].b$ has (so far) been changed from $false$ to $true$ only once (by process p at statement 3). Thus, no process other than p has (yet) exited from filter $A[i]$. Moreover, any process that subsequently exits from $A[i]$ will become blocked at the **await** of filter $A[i+1]$. Thus, p can elect itself as the leader. From the filter safety property, at most one process enters filter $A[\lceil \log_2 N \rceil]$, and hence $\Theta(\log N)$ filters suffice. Thus, the algorithm's space complexity is $\Theta(\log N)$.

Mutual Exclusion: an unbounded version. The election algorithm can be used to create a simple unbounded mutual exclusion algorithm. This algorithm is shown in Figure 2.6(b). The algorithm uses an unbounded number of filters, $A[0]$, $A[1]$, $\ldots$. All competing processes participate in the election algorithm using the set of filters $A[\mathit{Entry}]$, $A[\mathit{Entry}+1]$, $\ldots$, where *Entry* is a shared variable that "points" to the next unused filter. If a process is elected as the leader, then it enters its critical section. In its exit section, the leader initiates another round of the election algorithm by updating *Entry*. (Since each filter is properly initialized and used only once, a process only has to update *Entry* to point to a new filter.) The algorithm has $O(k')$ system response time. (The analysis of this is rather complicated and will not be presented here. We refer the interested reader to [30].) However, since a process may repeatedly lose in the election algorithm, the algorithm is not starvation-free.

The arbiter mechanism. The algorithm just described has two shortcomings: it uses unbounded memory, and is not starvation-free. Choy and Singh proposed an *arbiter* mechanism as a solution to both problems. In this mechanism, a shared variable *Arbt* is used to ensure that each process in its entry section eventually is accorded a chance to enter its critical section. The *Arbt* pointer is updated each time a critical section is executed, and cycles through the processes in a round-robin fashion. When a process p exits its critical section, it first checks whether the process pointed to by *Arbt* is in its entry section. If so, process p signals that process to immediately enter its critical section. The next election algorithm starts only after the process pointed to by *Arbt* finishes its critical section.

In this way, a process is guaranteed to enter its critical section after at most N critical-section executions. Since each election round uses $O(\log N)$ filters, and since no process is left in an old election round after N rounds, the number of filters can be limited to $\Theta(N \log N)$. Also, since each election algorithm uses $O(\log k')$ filters, where k' is the interval contention over the election period, and since each filter has $O(1)$ step time complexity, the entire algorithm has $O(N \log k')$ step time complexity.

Choy and Singh also presented further optimizations, which result in $O(k')$ system response time, $O(1)$ amortized system response time, and $O(N)$ step time complexity. Later, Attiya and Bortnikov [24] devised an improved filter algorithm, which has $O(k)$

```
shared variables
    $Number$: array[0..$N-1$] of $0..\infty$ initially 0;
    $Choosing$: array[0..$N-1$] of boolean initially $false$
process $p$                                   /* $0 \le p < N$ */
private variables
    $q$: $0..N-1$;
    $S$: (a set of processes)
while $true$ do
1:  Noncritical Section;
2:  Join($p$);
3:  $Choosing[p] := true$;
4:  $S :=$ Get_Set();
5:  $Number[p] := 1 + \max_{q \in S} Number[q]$;
6:  $Choosing[p] := false$;
7:  $S :=$ Get_Set();
8:  for each $q \in S - \{p\}$ do
9:       await $\neg Choosing[q]$;
10:      await $(Number[q] = 0) \vee$
                 $(Number[p], p) < (Number[q], q)$
    od;
11: Critical Section;
12: $Number[p] := 0$;
13: Leave($p$)
od
```

Figure 2.7: Algorithm AST: An adaptive bakery algorithm by Afek, Stupp, and Touitou [5].

step time complexity, $O(\log k)$ system response time, and $O(N \log M)$ space complexity, where k is point contention. They also presented another algorithm with $O(M \log M)$ space complexity, but at the price of adapting only to *interval* contention (k'). From these results, we have the following theorem.

Theorem 2.6 (Attiya and Bortnikov) *The mutual exclusion problem can be solved with $O(\log k)$ system response time and $O(k)$ step time complexity using only reads and writes, where k is point contention.* □

2.4.2 An Adaptive Bakery Algorithm

Afek, Stupp, and Touitou [5] constructed an adaptive bakery algorithm, by combining Lamport's bakery algorithm [51] with a wait-free *active set* object. Their algorithm, denoted Algorithm AST, is illustrated in Figure 2.7. An active set object manages a set of processes, and each process p may execute the following three operations.

- Join(p): adds p to the active set.
- Leave(p): removes p from the active set.
- Get_Set(): returns the current active set S. If a process q's execution of Join(q) or Leave(q) overlaps p's execution of Get_Set, then S is allowed to either contain q or not.

Afek, *et al.* also devised a wait-free implementation of an active set object based on a generic adaptive long-lived renaming algorithm. Several such algorithms are presented in [1, 2], with various tradeoffs among the following parameters: the size of the output name space, step time complexity, space complexity, and whether the algorithm is adaptive to interval or point contention. By using an $O(k^2)$-renaming algorithm presented in [1], the active set algorithm of Afek, *et al.* achieves $O(k^4)$ step time complexity, where k is point contention.

If we remove statements 2 and 13 of Algorithm AST and change statements 4 and 7 to $S := \{0..N-1\}$, then we obtain the original bakery algorithm. Due to these four statements, Algorithm AST only has to check processes that are concurrently active, instead of checking every process in the system. Thus, the **for** loop of statements 8–10 has $O(k)$ step time complexity.

Clearly, overall time complexity is dominated by the active set algorithm, and hence Algorithm AST has $O(k^4)$ step time complexity. Since the active set algorithm is wait-free, it can be shown that Algorithm AST also has $O(k^4)$ system response time. Unfortunately, as in the original bakery algorithm, each process's *Number* variable in Algorithm AST has unbounded range. (The total number of shared variables is $\Theta(N^3 + M^3 N)$.)

2.4.3 Adaptive Mutual Exclusion with Infinitely Many Processes

Merritt and Taubenfeld [64] investigated mutual exclusion, as well as other concurrent programming problems, when the number of participating processes is possibly unbounded. In particular, they presented a number of mutual exclusion algorithms under various system models, some of which are adaptive under the system-response-time measure. From their results, we have the following theorem.

Theorem 2.7 (Merritt and Taubenfeld) *For systems with a possibly unbounded number of processes, there exists a starvation-free mutual exclusion algorithm with infinitely many variables and system response time that is a function of interval contention.* □

2.4.4 Local-spin Adaptive Algorithms

None of the previously-cited adaptive algorithms is a local-spin algorithm, and thus each has unbounded RMR time complexity. Surprisingly, while adaptivity and local spinning have been the predominant themes in recent work on mutual exclusion, the problem of designing an adaptive, local-spin algorithm under read/write atomicity has remained open until recently. In Chapter 4, we close this problem by presenting an adaptive algorithm with $\Theta(\min(k, \log N))$ RMR time complexity, where k is point contention. This algorithm has $\Theta(N)$ space complexity, which is clearly optimal.

In addition, Afek, Stupp, and Touitou [6] have independently devised another local-spin adaptive algorithm, based on a long-lived implementation of a splitter element, with a structure that is similar to our algorithm. Due to the close similarity of these two algorithms, the algorithm of Afek *et al.* is considered in Chapter 4.

2.5 Mutual Exclusion with Nonatomic Operations

To this point, we have assumed that shared variables are accessed atomically. Requiring atomic variable accesses is tantamount to assuming mutual exclusion at some lower level. Thus, mutual exclusion algorithms requiring this are in some sense circular. Lamport recognized that mutual exclusion can be implemented without requiring operations to be atomic in some early papers [51, 52], and later wrote more extensively on the topic in a two-part work [53, 54].

One shortcoming of Lamport's algorithms is that they are not local-spin algorithms. In later work, Anderson devised a nonatomic algorithm in which all spins are local [11]. As mentioned previously, this algorithm was actually the first local-spin algorithm that uses only read and write operations. This algorithm, hereafter referred to as Algorithm JA, is described below.

Algorithm JA was obtained by first solving the two-process case, and by then using the two-process solution to solve the N-process case. The two-process version of Algorithm JA is shown in Figure 2.8. The two processes are denoted u and v. With

```
shared variable $P$, $Q$, $T$: array$[u, v]$ of boolean initially $true$

process $u$                                   process $v$

private variable $x$: boolean                 private variable $x$: boolean
initially $x = T[u]$                          initially $x = T[v]$

while $true$ do                               while $true$ do
     1:  Noncritical Section;                      1:  Noncritical Section;
     2:  $P[u] := false$;                          2:  $P[v] := false$;
     3:  $Q[u] := false$;                          3:  $Q[v] := false$;
     4:  $x := T[v]$;                              4:  $x := \neg T[u]$;
     5:  $T[u] := x$;                              5:  $T[v] := x$;
     6:  if $x$ then                               6:  if $x$ then
     7:      $P[u] := true$;                       7:      $Q[v] := true$;
     8:      await $P[v]$                          8:      await $P[u]$
         else                                          else
     9:      $Q[u] := true$;                       9:      $P[v] := true$;
    10:      await $Q[v]$                         10:      await $Q[u]$
         fi;                                           fi;
    11:  Critical Section;                        11:  Critical Section;
    12:  $P[u] := true$;                          12:  $P[v] := true$;
    13:  $Q[u] := true$                           13:  $Q[v] := true$
od                                            od
```

Figure 2.8: Algorithm JA: Two-process case.

the exception of statements 4, 7, and 9, the two processes are identical. Algorithm JA is similar to earlier two-process solutions given by Peterson [69] and by Kessels [45], but uses only single-reader, single-writer boolean variables.

The two shared variables $T[u]$ and $T[v]$ are used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. Process u attempts to establish $T[u] = T[v]$ and process v attempts to establish $T[u] \neq T[v]$. Process u enters its critical section only if $T[u] \neq T[v]$ holds or if process v has not expressed interest in entering its critical section. Similarly, process v enters its critical section only if $T[u] = T[v]$ holds or if process u has not expressed interest in entering its critical section. Thus, in the event of a "tie," process u is favored if $T[u]$ and $T[v]$ differ in value and process v is favored if $T[u]$ and $T[v]$ are equal. This is essentially the idea of Kessels' algorithm. In Algorithm JA, each process checks the condition that is required for it to enter its critical section by waiting on one single-reader, single-writer boolean variable. The manner in which this is accomplished is explained next.

Because $T[u]$ is written only by process u, process u can keep track of its value by using a private variable; variable $u.x$ is used for this purpose. In order for process u to wait until $T[u] \neq T[v]$ holds, it simply tests $u.x$ and then waits for $T[v]$ to have the appropriate value. In particular, if $u.x$ is true (which implies that $T[u]$ is *true*), then process u waits until $T[v]$ is *false*, and if $u.x$ is *false* (which implies that $T[u]$ is *false*), then process u waits until $T[v]$ is *true*. Process u waits for $T[v]$ to have the appropriate value by waiting on either $P[v]$ or $Q[v]$. As explained next, these variables serve the dual purpose of "signaling" the value of $T[v]$ and "signaling" that process v is in its noncritical section.

Loosely speaking, $P[v]$ is used by process v to signal to process u that $T[v]$ is *false*, and $Q[v]$ is similarly used to signal that $T[v]$ is *true*. While the value of $T[v]$ is being determined in statements 4 and 5 of process v, the appropriate value to signal is not known; thus, to ensure that process u does not enter its critical section when it should not, $P[v]$ and $Q[v]$ are both kept equal to *false* while this value is being determined. When process v is in its noncritical section (where it may halt), process u should not be blocked in its entry section; thus, while v is in its noncritical section, $P[v]$ and $Q[v]$ are both kept equal to *true*. In this way, $P[v]$ and $Q[v]$ serve both purposes mentioned in the previous paragraph. Variables $P[u]$ and $Q[u]$ are similarly used by process u to signal the value of $T[u]$ to process v, except their roles are reversed: $P[u]$ is used to signal that $T[u]$ is *true*, and $Q[u]$ is used to signal that $T[u]$ is *false*.

The N-process version of ALGORITHM JA is obtained by applying the two-process version in the following.

```
process p      /* $0 \leq p < N$ */
while true do
    Noncritical Section;
    for $i := 0$ to $N - 1$ do
        if $i \neq p$ then ENTRY($p$, $i$)
    od;
    Critical Section;
    for $i := 0$ to $N - 1$ do
        if $i \neq p$ then EXIT($p$, $i$)
    od
od
```

In this algorithm, `ENTRY`(i, j) denotes the entry section from a two-process solution that process i executes to compete with process j. `EXIT`(i, j) is defined similarly. `ENTRY`(i, j) and `EXIT`(i, j) are assumed to be implemented using the algorithm in Figure 2.8. It should be straightforward to see that the two-process version of ALGORITHM JA has $O(1)$ RMR time complexity, and the N-process version has $\Theta(N)$ RMR time complexity.

As mentioned above, ALGORITHM JA remains correct even if variable accesses are nonatomic. Since only single-reader, single-writer boolean variables are used in the algorithm, the only potential problem that must be considered is the case of a read of a boolean variable that overlaps a write of that variable. In such a case, it can be shown that it is safe for the read to return either *true* or *false*. Thus, we have the following theorem.

Theorem 2.8 (Anderson) *The mutual exclusion problem can be solved with* $\Theta(N)$ *RMR time complexity using only nonatomic reads and writes under either the CC or DSM model.* $\Box$

In Chapter 7, we show that by applying simple transformations to ALGORITHM YA-N, covered earlier in Section 2.2.2, a nonatomic algorithm with the same RMR time complexity can be derived. The key idea is to replace all multi-reader, multi-writer, multi-bit variables by single-reader, single-writer, single-bit variables, so that overlapping operations on the same variable have no adverse impact.

2.6 Time- and Space-complexity Lower Bounds

In this section, we survey several papers that present time- and space-complexity lower bounds for mutual exclusion. We begin in Section 2.6.1 by briefly discussing some of the proof techniques used in these papers. Then, in Section 2.6.2, a summary of results is presented.

2.6.1 Overview of Proof Methodologies

Most interesting lower bounds pertaining to the mutual exclusion problem are based on arguments that are quite complex. Therefore, we provide only a brief overview of commonly-used proof techniques.

Time-complexity lower bounds for mutual exclusion are often derived by inductively constructing longer and longer histories. Usually, such histories must be constructed in a way that limits "information flow" among competing processes. Information flow occurs when one process reads (from a shared variable) a value that was written by another process. If n processes are in their entry sections, and no information flow among them has occurred, then at least $n-1$ of them must perform further accesses to shared variables; otherwise, the current history under consideration can be extended to one in which multiple processes are in their critical sections. In other words, if no information flow has occurred among processes in their entry sections, then "most" of these processes have a "next" shared-variable access. By inductively appending such variable accesses while limiting information flow, a lower bound on entry-section execution time can be derived.

One way to prevent information flow is by "erasing" processes in the history under consideration [22, 33]. When a process is erased, its statement executions are completely removed from the history. For example, suppose a history is to be extended by appending the next statement execution of each process in its entry section. If the statement to be appended for process p is a read of some variable x that was previously written, and process q was the last process to write x, then the resulting information flow from q to p can be eliminated by erasing either p or q. Of course, if another process wrote x prior to q, then erasing q does not eliminate all information flow to p. In general, determining which processes to erase so that the induction can continue is a tricky balancing act.

Sometimes, erasing alone does not leave enough processes for the next induction step. In this case, the "roll-forward" strategy can be used: some processes are selected and allowed to "roll forward" until they return to their noncritical sections [33]. For instance, in the example above, information flow among *competing* processes can be eliminated by allowing q to roll forward. In this way, p's read of x obtains a value written by a process in its noncritical section, *i.e.*, a process that p is not competing with. Of course, as q is rolled forward, it may read shared variables that have been written by other processes. Depending on the proof strategy being used, the resulting information flow may need to be dealt with by erasing or rolling forward other processes. These two strategies, erasing and roll-forward, are used in the lower-bound proofs presented in Chapters 5 and 6.

Note that, in the example in the previous paragraph, q's write to x *eliminates* any information flow through x among concurrently-competing processes. This basic proof technique, in which certain processes may be "hidden" by the writes of other processes, dates back to a paper on the space complexity of mutual exclusion by Burns and Lynch [28, 29].

2.6.2 Lower-bound Results

We now present an overview of research on lower bounds.

Alur and Taubenfeld proved that a naive definition of "time complexity" (in which *every* shared-variable access is counted) is not meaningful for mutual exclusion [8].

Theorem 2.9 (Alur and Taubenfeld) *For any N-process mutual exclusion algorithm with $N \geq 2$, the first process to enter its critical section may perform an unbounded number of shared-variable accesses.* $\square$

Anderson and Yang presented several lower bounds that establish trade-offs between the number of remote memory references required for mutual exclusion and write- and access-contention [22]. The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may potentially be simultaneously enabled to write (access) the same shared variable.[3]

Theorem 2.10 (Anderson and Yang) *For any N-process mutual exclusion algorithm, if write-contention is w, and if each atomic operation accesses at most v remote variables, then there exists a history involving only one process in which that process performs $\Omega(\log_{vw} N)$ remote memory references (under the DSM model) for entry into its critical section. Moreover, among these memory references, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed.* $\square$

Theorem 2.11 (Anderson and Yang) *For any N-process mutual exclusion algorithm, if access-contention is c, and if each atomic operation accesses at most v remote variables, then there exists a history involving only one process in which that process accesses $\Omega(\log_{vc} N)$ distinct remote variables for entry into its critical section.* $\square$

According to Theorem 2.10, under the DSM model, $\Omega(\log_{vw} N)$ remote memory references are required, even in the absence of contention. In CC machines, the first access of a remote variable must cause a cache miss. Thus, by counting the number of distinct remote variables accessed, a lower bound for the CC model is obtained. Thus, Theorems 2.10 and 2.11 imply that $\Omega(\sqrt{\log_{vw} N})$ (or $\Omega(\log_{vc} N)$) remote memory references are required under the CC model in the absence of contention. Note that Theorem 2.10 implies that fast mutual exclusion algorithms require arbitrarily high write-contention. Thus, the fact that Algorithm L uses N-writer shared variables is no accident.

Burns and Lynch considered the space complexity of mutual exclusion algorithms [29].

[3]The notions of write-contention, access-contention, and contention cost (defined later in this section) are different from the concepts of point and interval contention defined earlier in Section 2.4.

Theorem 2.12 (Burns and Lynch) *Any livelock-free N-process mutual exclusion algorithm that uses only reads and writes must use at least N shared variables.* □

They also showed that this bound is tight by presenting a livelock-free algorithm with N boolean shared variables.

Alur and Taubenfeld investigated limitations on variable size in algorithms that use only atomic reads and writes [9].

Theorem 2.13 (Alur and Taubenfeld) *For any N-process mutual exclusion algorithm using only reads and writes, if each shared variable has l bits, then there exists a history involving only one process in which that process executes at least $\big((\log_2 N)/(l - 2 + 3\log_2\log_2 N)\big)$ operations for entry into its critical section. Moreover, among these operations, at least $\sqrt{(\log_2 N)/(l + \log_2\log_2 N)}$ distinct shared variables are accessed.* □

They also established the following upper bound by constructing a simple variant of Algorithm L.

Theorem 2.14 (Alur and Taubenfeld) *For every l such that $1 \leq l \leq \log N$, there exists a livelock-free N-process mutual exclusion algorithm using only reads and writes, where each shared variable has l bits, such that a process executes at most $7\lceil(\log_2 N)/l\rceil$ operations to enter and exit its critical section in the absence of contention. Moreover, among these operations, at most $3\lceil(\log_2 N)/l\rceil$ different shared variables are accessed.* □

According to Theorem 2.13, a "fast" algorithm requires variables with $\Omega(\log N)$ bits, *i.e.*, variables large enough to hold process identifiers, or at least some constant fraction of a process identifier. Most modern multiprocessor systems have at least 32-bit memory words, so requiring variables of size $\Omega(\log N)$ usually poses no problem in practice. However, there exist systems and algorithms that exploit the ability to access memory at different granularities. For example, as mentioned in Section 2.3.1, Michael and Scott presented a variant of Algorithm L in which processes access memory at both full- and half-word granularities [65]. Alur and Taubenfeld's results show that there are limitations to this approach.

Dwork, Herlihy, and Waarts questioned the assumption that each access to a variable is equally expensive [35]. Their chief contribution was to introduce a formal complexity model that takes into account the (hardware) contention caused by overlapping accesses to the same variable. Specifically, their model distinguishes between the *invocation* and *response* of each operation. An operation's *contention cost* is defined to be the number of response events from the same variable that occur between the operation's invocation and matching response. Note that the access contention of a program, as defined earlier, is simply the worst-case contention cost of any variable. Dwork, *et al.* applied this model to various classes of algorithms in order to study trade-offs between average/worst-case contention cost and time complexity. Regarding the mutual exclusion problem, they proved the following.

Theorem 2.15 (Dwork, *et al.*) *For any N-process mutual exclusion algorithm using reads, writes, and read-modify-write operations, if access-contention is c, then there exists a history in which a process executes $\Omega((\log N)/c)$ operations for entry into its critical section.* □

Note that Theorems 2.13 and 2.15 do not distinguish between local and remote memory references.

Cypher [33] was the first to present a lower bound on remote memory references under arbitrary access-contention. His lower bound applies to algorithms using reads, writes, and comparison primitives such as *compare-and-swap*, and *test-and-set*.

Theorem 2.16 (Cypher) *For any N-process mutual exclusion algorithm using reads, writes, and comparison primitives, there exists a history satisfying the following:*

$$\frac{\textit{the total number of remote memory references}}{\textit{the number of processes that participate in the history}} = \Omega(\log\log N/\log\log\log N).$$

□

Cypher's lower bound applies to either DSM or CC systems. Given that primitives such as *fetch-and-increment* and *fetch-and-store* can be used to implement mutual exclusion in $O(1)$ time, Cypher's result pointed to an unexpected weakness of *compare-and-swap*, which is still widely regarded as being the most useful of all primitives to provide in hardware.

As noted earlier, in Chapter 5, we improve Cypher's result to obtain a lower bound of $\Omega(\log N/\log\log N)$. This lower bound also applies to both CC and DSM systems.

Also, in Chapter 6, we present a time-complexity lower bound for adaptive mutual exclusion. This lower bound precludes the possibility of an $o(k)$ adaptive algorithm. Finally, in Chapter 7, we present a time-complexity lower bound of $\Omega(\log N/\log\log N)$

for nonatomic mutual exclusion, which precludes the possibility of a fast or adaptive nonatomic algorithm. (See Section 1.4.)

CHAPTER 3

Space-optimal Mutual Exclusion Under Read/Write Atomicity*

In this chapter, a simple code transformation is presented that reduces the space complexity of Yang and Anderson's local-spin mutual exclusion algorithm, which was described in Section 2.2.2 (Algorithm YA-N). In both the original and the transformed algorithm, only atomic read and write instructions are used; each process generates $\Theta(\log N)$ RMRs (remote memory references) per lock request, where N is the number of processes. The transformed algorithm uses $\Theta(N)$ distinct variables, which is clearly optimal. This algorithm is used in Chapter 4 to construct a space-optimal adaptive mutual exclusion algorithm with $\Theta(\min(k, \log N))$ RMR time complexity, where k is point contention.

As described in Section 2.2.2, Algorithm YA-N is constructed by embedding instances of a two-process mutual exclusion algorithm (Algorithm YA-2) within a binary arbitration tree. The two-process algorithm has $O(1)$ time complexity, so the overall per-process RMR time complexity is $\Theta(\log N)$. The algorithm's correctness relies crucially on the fact that, when a process p is blocked within a node n of the tree, it can be later released only by the (unique) "winning" process at that node — in particular, other processes that p may compete with elsewhere in the tree should not "interfere" with the node-n algorithm by updating p's spin variable at that node. To prevent such interference, a distinct spin variable is used for each process at each level of the tree. Thus, the algorithm's space complexity is $\Theta(N \log N)$ (where, as noted earlier, "space complexity" is defined as the total number of variables used).

*The results presented in this chapter have been published in the following paper.
[47] Y.-J. Kim and J. Anderson. A space- and time-efficient local-spin spin lock. *Information Processing Letters*, 84(1):47–55, September 2002.

```
const                                              /* for simplicity, we assume $N = 2^L$ */
    $L = \log N$;                                  /* height of arbitration tree = $O(\log N)$ */
    $Tsize = 2^L - 1 = N - 1$                      /* size of arbitration tree = $O(N)$ */

shared variables
    $T$: array[1..$Tsize$] of $0..N-1$;
    $C$: array[1..$Tsize$][0..1] of $(0..N-1, \perp)$ initially $\perp$;
    $R$: array[1..$Tsize$][0..1] of $(0..N-1, \perp)$;               /* ALGORITHM LS */
    $Q$: array[1..$L$][0..$N-1$] of 0..2 initially 0;                 /* ALGORITHM YA-N */
    $P$: array[1..$Tsize$][0..1] of 0..2 initially 0;           /* ALGORITHMS CC, LS and F */
    $S$: array[0..$N-1$] of boolean initially false                /* ALGORITHMS LS and F */

private variables
    $h$: 1..$L$;
    $node$: 1..$Tsize$;
    $side$: 0..1;                                          /* 0 = left side, 1 = right side */
    $rival$: $0..N-1$
```

Figure 3.1: Variable declarations.

In this chapter, we present a simple code transformation that reduces the space complexity of Algorithm YA-N from $\Theta(N \log N)$ to $\Theta(N)$. In our new algorithm, each process uses the same spin variable for all levels of the arbitration tree. The algorithm is constructed in a way that prevents interference between different nodes from having any ill effect. Like Algorithm YA-N, the RMR time complexity of our new algorithm is $\Theta(\log N)$.

This chapter is organized as follows. In Section 3.1, we show that our new algorithm can be derived from Algorithm YA-N in a manner that preserves the Exclusion and Starvation-freedom properties. In Section 3.2, we prove that the algorithm's RMR time complexity is $\Theta(\log N)$. We conclude in Section 3.3.

3.1 Transformation of Algorithm YA-N

Starting with Algorithm YA-N, we construct three other algorithms, each obtained from its predecessor by means of a simple code transformation. These other algorithms are Algorithm CC (for cache-coherent), Algorithm LS (for linear space), and Algorithm F (the final algorithm). The first two algorithms are shown in Figure 3.2, and the second two in Figure 3.3. Variable declarations for all the algorithms are given in Figure 3.1.

/$*$ statements of Algorithm CC that are different from Algorithm YA-N have **boldface** line numbers $*$/

Algorithm YA-N
/$*$ the original algorithm in [84] $*$/

```
    process $p$ ::    /$*$ $0 \le p < N$ $*$/
    while $true$ do
1:  Noncritical Section;

    for $h := 1$ to $L$ do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
2:      $C[node][side] := p$;
3:      $T[node] := p$;
4:      $Q[h][p] := 0$;
5:      $rival := C[node][1 - side]$;
        if ($rival \ne \perp$ $\wedge$
6:            $T[node] = p$) then
7:          if $Q[h][rival] = 0$ then
8:              $Q[h][rival] := 1$
            fi;
9:          await $Q[h][p] \ge 1$;
10:         if $T[node] = p$ then
11:             await $Q[h][p] = 2$
            fi
        fi
    od;

12: Critical Section;

    for $h := L$ downto $1$ do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
13:     $C[node][side] := \perp$;
14:     $rival := T[node]$;
        if $rival \ne p$ then
15:         $Q[h][rival] := 2$
        fi
    od
od
```

Algorithm CC
/$*$ the cache-coherent algorithm $*$/

```
    process $p$ ::    /$*$ $0 \le p < N$ $*$/
    while $true$ do
1:  Noncritical Section;

    for $h := 1$ to $L$ do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
2:      $C[node][side] := p$;
3:      $T[node] := p$;
4:      $P[node][side] := 0$;
5:      $rival := C[node][1 - side]$;
        if ($rival \ne \perp$ $\wedge$
6:            $T[node] = p$) then
7:          if $P[node][1 - side] = 0$ then
8:              $P[node][1 - side] := 1$
            fi;
9:          await $P[node][side] \ge 1$;
10:         if $T[node] = p$ then
11:             await $P[node][side] = 2$
            fi
        fi
    od;

12: Critical Section;

    for $h := L$ downto $1$ do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
13:     $C[node][side] := \perp$;
14:     $rival := T[node]$;
        if $rival \ne p$ then
15:         $P[node][1 - side] := 2$
        fi
    od
od
```

Figure 3.2: Algorithm YA-N and Algorithm CC.

Algorithm YA-N. We begin with a brief, informal description of Algorithm YA-N. At each node n at height h in the arbitration tree, the following variables are used: $C[n][0]$, $C[n][1]$, $T[n]$, and a subset of $Q[h][0], \ldots, Q[h][N-1]$. Variable $C[n][0]$ ranges over $\{0, \ldots, N-1, \perp\}$ and is used by a process from the left subtree rooted at n to inform a process from the right subtree rooted at n of its intent to enter its critical section. Variable $C[n][1]$ is similarly used by processes from the right subtree. Variable $T[n]$ ranges over $\{0, \ldots, N-1\}$ and is used as a tie-breaker in the event that two processes attempt to "acquire" node n at the same time. In such a case, the process that first updates $T[n]$ is favored. Variable $Q[h][p]$ is the spin variable used by process p at node n (if it is among the processes that, by the structure of the tree, can access node n). At each node n, the two-process algorithm behaves as explained in Section 2.2.2, except that variables $C[u]$, $C[v]$, T, $P[u]$, and $P[v]$ (shown in Figure 2.3) are replaced by $C[n][0]$, $C[n][1]$, $T[n]$, $Q[h][l]$, $Q[h][r]$, respectively (where l (r) is the process from the left (right) subtree).

Algorithm CC. In Algorithm CC, each node n has two associated spin variables, $P[n][0]$ and $P[n][1]$. $P[n][0]$ is used by *all* processes that try to acquire node n from the left side. $P[n][1]$ is similarly used by all right-side processes. Algorithm CC uses three shared arrays, C, P, and T, each with $\Theta(N)$ elements. Thus, its space complexity is $\Theta(N)$. Its RMR time complexity remains $\Theta(\log N)$ on cache-coherent (CC) systems, because each spin variable is waited on by at most one process at any time. For example, when a left-side process p repeatedly reads $P[n][0]$ at statement 9, the first such read causes $P[n][0]$ migrate to p's local cache; any subsequent reads before $P[n][0]$ is written are therefore local. The algorithm ensures that once $P[n][0]$ is written, p's busy-waiting loop terminates.

In contrast, Algorithm CC has unbounded time complexity on DSM systems without coherent caches. This is because different processes may block on the same spin variable at different times, which makes it impossible to statically allocate spin variables to processes in such a way that all spins are local.

Note that the two-process version of Algorithm CC is isomorphic to Algorithm YA-2. Thus, the correctness of the two-process version of Algorithm CC follows directly from the correctness of Algorithm YA-2. Because the two-process version of Algorithm CC is correct, its application within an N-process arbitration tree results in a correct N-process algorithm. (The arbitration-tree approach can be applied using *any* correct two-process algorithm, as long as the instances of the two-process algorithm used employ distinct variables.) As it turns out, Algorithm YA-N is actually trickier to prove correct than Algorithm CC. This is because, in Algorithm YA-N, the basic two-process version has been modified in a way that ensures that different processes use different spin variables at each node. In Algorithm CC, the two-process algorithm is being applied directly without modification.

Algorithm LS. Algorithm LS has been obtained from Algorithm CC by applying a simple transformation, which we examine here in isolation. In Algorithm

CC, all busy-waiting is by means of statements of the form "**await** B," where B is some boolean condition. Moreover, if a process p is waiting for condition B to hold, then there is a unique process that can establish B, and once B is established, it remains true, until p's "**await** B" statement terminates.

In Algorithm LS, each statement of the form "**await** B" has been replaced by the following code fragment:

```
a:  $R := p$;
b:  while $\neg B$ do
c:          await $S[p]$;
d:          $S[p] := \mathit{false}$
    od
```

where $S[p]$ is initially *false* (see statements 9 and 11). In addition, each assignment of the form "$B := \mathit{true}$" has been replaced by the following:

```
e:  $B := \mathit{true}$;
f:  $\mathit{rival} := R$;
g:  $S[\mathit{rival}] := \mathit{true}$
```

(see statements 8 and 15). Note that the code implementing "**await** B" can terminate only if B is *true*, *i.e.*, it terminates only when it should. Moreover, if a process p finds that B is *false* at statement b, and if another process q subsequently establishes B by executing statement e, then because p's execution of statement a precedes q's execution of statement f, q establishes $S[p] = \mathit{true}$ when it executes statement g. Thus, if a process p is waiting for condition B to hold, and B is established by another process, then p must eventually exit the **while** loop at statement b.

Algorithm F. Algorithm F has been obtained from Algorithm LS by removing the shared array R, which was introduced in applying the transformation above at each node of the arbitration tree. The fact that this array is unnecessary follows from several invariants of Algorithm LS, which are stated below. In stating these invariants (as well as those needed in later chapters), we use the following notational conventions.

Notational conventions. We use $s.p$ to denote the statement with label s of process p, and (as in Chapter 2) $p.v$ to represent p's private variable v. Let S be a subset of the statement labels in process p. Then, $p@S$ holds if and only if the program counter for process p equals some value in S.

/∗ statements that are different from ALGORITHM CC have with **boldface** line numbers ∗/

ALGORITHM LS
/∗ the linear-space algorithm ∗/

```
process $p$ ::    /∗ $0 \le p < N$ ∗/
while $true$ do
1:   Noncritical Section;
     for $h := 1$ to $L$ do
          $node := \lfloor (N+p)/2^h \rfloor$;
          $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
2:        $C[node][side] := p$;
3:        $T[node] := p$;
4:        $P[node][side] := 0$;
5:        $rival := C[node][1-side]$;
          if ($rival \neq \perp$ ∧
6:                 $T[node] = p$) then
7:             if $P[node][1-side] = 0$ then
8:   8e             $P[node][1-side] := 1$;
     8f             $rival := R[node][1-side]$;
     8g             $S[rival] := true$
               fi;
9:   9a        $R[node][side] := p$;
     9b        while $P[node][side] = 0$ do
     9c             await $S[p]$;
     9d             $S[p] := false$
               od;
10:            if $T[node] = p$ then
11:  11a            $R[node][side] := p$;
     11b            while $P[node][side] \le 1$ do
     11c                 await $S[p]$;
     11d                 $S[p] := false$
                    od
               fi
          fi
     od;
12:  Critical Section;
     for $h := L$ downto $1$ do
          $node := \lfloor (N+p)/2^h \rfloor$;
          $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
13:       $C[node][side] := \perp$;
14:       $rival := T[node]$;
          if $rival \neq p$ then
15:  15e       $P[node][1-side] := 2$;
     15f       $rival := R[node][1-side]$;
     15g       $S[rival] := true$
          fi
     od
od
```

ALGORITHM F
/∗ the final algorithm ∗/

```
process $p$ ::    /∗ $0 \le p < N$ ∗/
while $true$ do
1:   Noncritical Section;
     for $h := 1$ to $L$ do
          $node := \lfloor (N+p)/2^h \rfloor$;
          $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
2:        $C[node][side] := p$;
3:        $T[node] := p$;
4:        $P[node][side] := 0$;
5:        $rival := C[node][1-side]$;
          if ($rival \neq \perp$ ∧
6:                 $T[node] = p$) then
7:             if $P[node][1-side] = 0$ then
8:   8e             $P[node][1-side] := 1$;
     8f               —
     8g             $S[rival] := true$
               fi;
9:   9a          —
     9b        while $P[node][side] = 0$ do
     9c             await $S[p]$;
     9d             $S[p] := false$
               od;
10:            if $T[node] = p$ then
11:  11a              —
     11b            while $P[node][side] \le 1$ do
     11c                 await $S[p]$;
     11d                 $S[p] := false$
                    od
               fi
          fi
     od;
12:  Critical Section;
     for $h := L$ downto $1$ do
          $node := \lfloor (N+p)/2^h \rfloor$;
          $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
13:       $C[node][side] := \perp$;
14:       $rival := T[node]$;
          if $rival \neq p$ then
15:  15e       $P[node][1-side] := 2$;
     15f         —
     15g       $S[rival] := true$
          fi
     od
od
```

Figure 3.3: ALGORITHM LS and ALGORITHM F.

(Note that if s is a statement label, then $p@\{s\}$ means that statement s of process p is *enabled*, *i.e.*, p has not yet executed s.)

We assume that each labeled sequence of statement(s) is atomic. Note that each numbered statement reads or writes at most one shared variable. $\square$

Before stating the required invariants, first note that removing array R does not affect safety, because the **while** conditions at statements 9b and 11b still ensure that the busy-waiting loops terminate only when they should. However, there is now a potential danger that a process may not correctly update its rival's spin variable, and hence that process may wait forever at either statement 9c or 11c. The following invariants (of Algorithm LS) imply that this is not possible.

Informally, invariant (I1) below implies that, as long as q is spinning at node n, the entries of R and C that pertain to node n correctly show q's presence. Invariant (I2) implies that, if p is at statements 6..8f and its variable $p.rival$ holds an inaccurate value (*i.e.*, $p.rival \neq C[p.node][1 - p.side]$), then p's true rival, q, either has not yet updated $T[p.node]$ (*i.e.*, $q@\{3\}$ holds), or has updated $T[p.node]$ and in fact did so after p (*i.e.*, $T[p.node] = q$ holds). In either case, because ties are broken in favor of p, it does not matter that $p.rival$ is inaccurate. Invariant (I3) implies that, if p is at statement 8f and its rival q is spinning, then either $p.rival$ is accurate (*i.e.*, $p.rival = R[p.node][1-p.side]$), or ties are broken in favor of p (*i.e.*, $T[p.node] = q$). Finally, (I4) implies that when p is about to exit from node n, if its rival, q, is still spinning, then $T[n]$ cannot hold the identifier of a process other than p or q. Therefore, p can correctly infer its competitor by reading $T[n]$.

invariant $q@\{\text{9b..9d}, \text{11b..11d}\} \Rightarrow$
$$R[q.node][q.side] = q \ \wedge \ C[q.node][q.side] = q \qquad \text{(I1)}$$

invariant $p@\{6..\text{8f}\} \ \wedge \ p.rival \neq C[p.node][1 - p.side] \ \wedge$
$q@\{3..\text{11d}\} \ \wedge \ q.node = p.node \ \wedge \ q.side = 1 - p.side \ \Rightarrow$
$$q@\{3\} \ \vee \ T[p.node] = q \qquad \text{(I2)}$$

invariant $p@\{\text{8f}\} \ \wedge \ q@\{\text{9b..9d}, \text{11b..11d}\} \ \wedge$
$q.node = p.node \ \wedge \ q.side = 1 - p.side \ \Rightarrow$
$$p.rival = R[p.node][1 - p.side] \ \vee \ T[p.node] = q \qquad \text{(I3)}$$

invariant $p@\{14, \text{15e..15g}\} \ \wedge \ q@\{\text{9b..9d}, \text{11b..11d}\} \ \wedge \ q.node = p.node \ \Rightarrow$
$$T[p.node] = p \ \vee \ T[p.node] = q \qquad \text{(I4)}$$

Invariant (I1) follows easily from the correctness of the arbitration-tree mechanism used in ALGORITHM LS and the fact that process q establishes $C[q.node][q.side] = q$ (statement 2) before it establishes $R[q.node][q.side] = q$ (statements 9a and 11a).

To see that (I2) holds, note that $p.rival = C[p.node][1-p.side]$ holds when $p@\{6\}$ is established by process p. The only other statements that may establish its antecedent are statements 2 and 13 of some other process q with $q.node = p.node \ \wedge\ q.side = 1-p.side$. These statements may establish $q@\{3..\mathtt{11d}\}$ or falsify $p.rival = C[p.node][1-p.side]$. However, $q@\{3..\mathtt{11d}\}$ is false after q executes statement 13 (moreover, by the correctness of the arbitration-tree mechanism, $r@\{3..\mathtt{11d}\} \ \wedge\ r.node = p.node \ \wedge\ r.side = 1-p.side$ is false after the execution of statement 13 by q for *any* choice of r). Also, statement 2 establishes $q@\{3\}$, *i.e.*, the consequent of (I2) holds. Finally, note that whenever $q@\{3\}$ is falsified, $T[p.node] = q$ is established.

Invariant (I3) follows from (I1) and (I2) by considering two cases. Assume that its antecedent holds. If $p.rival = C[p.node][1-p.side]$ holds, then $p.rival = R[p.node][1-p.side]$ follows by (I1). Otherwise, $T[p.node] = q$ follows by (I2).

By the correctness of the arbitration-tree mechanism, it easily follows that after both p and q have executed statement 3 with $p.node = n \ \wedge\ q.node = n$ and before either of them leaves node n, no other process can enter node n. Therefore, the consequent of (I4) is not falsified while its antecedent holds.

Our objective now is to show that the reads of $R[p.node][1-p.side]$ by p at statements 8f and 15f are superfluous. We show this by proving that if p's rival q is executing within statements 9b–9d or 11b–11d, where it may potentially block, and if p attempts to release its rival by executing statements 7–8g or statements 14–15g, then q cannot be blocked forever. Consider the following two lemmas.

Lemma 3.1 *If $q.node = p.node \ \wedge\ q.side = 1-p.side \ \wedge\ q@\{\mathtt{9b}..\mathtt{9d}, \mathtt{11b}..\mathtt{11d}\}$ holds while p executes statements 14–15f, then p's execution of statement 15f does not change the value of $p.rival$.*

Proof: By (I4), p must have found $T[p.node] = q$ at statement 14. By (I1), p also finds $R[p.node][1-p.side] = q$ at statement 15f. □

Of course, if p reaches statement 15f when $q.node = p.node \ \wedge\ q.side = 1-p.side$ holds and $q@\{\mathtt{9b}..\mathtt{9d}, \mathtt{11b}..\mathtt{11d}\}$ is established *afterwards*, then q executes statement 9b or 11b after p executes 15e, so q will not spin on variable $S[q]$. In other words, the potential removal of statement 15f causes no problems in this case.

Lemma 3.2 If $q.node = p.node \ \wedge \ q.side = 1 - p.side \ \wedge \ q@\{\mathsf{9b..9d}, \mathsf{11b..11d}\}$ holds while p executes statements 7–8f, then **(i)** p's execution of statement 8f does not change the value of $p.rival$, or **(ii)** p enters its critical section before q does.

Proof: By (I3), either $p.rival = R[p.node][1 - p.side]$ holds or $T[p.node] = q$ holds when p executes statement 8f. In the former case, statement 8f does not change the value of $p.rival$. In the latter case, by the tie-breaking strategy used in ALGORITHM LS, p enters its critical section before q. □

As before, the potential removal of statement 8f causes no problems if p reaches statement 8f when $q.node = p.node \wedge q.side = 1 - p.side$ holds and $q@\{\mathsf{9b..9d}, \mathsf{11b..11d}\}$ is established *afterwards*. Note that if (ii) above applies, then because p enters its critical section before q, p will eventually execute statements 15e–15g and release q from its spinning. Hence, the algorithm remains starvation-free if both statements 8f and 15f are removed. Finally, with statements 8f and 15f removed, statements 9a and 11a can be eliminated as well. Thus, we have the following theorem.

Theorem 3.1 ALGORITHM F *is a correct, starvation-free mutual exclusion algorithm, with* $\Theta(N)$ *space complexity.* □

3.2 Time Complexity

Although we showed in the previous section that ALGORITHM F is a starvation-free algorithm, we did not establish its time complexity. (Starvation-freedom merely indicates that a process *eventually* enters its critical section; it does not specify how soon.) In this section, we show that each process performs $\Theta(\log N)$ remote memory references in ALGORITHM F to enter and then exit its critical section. Because the spins at statements 9c and 11c are local, it clearly suffices to establish a $\Theta(\log N)$ bound on the total number of iterations of the **while** loops at statements 9b and 11b for one complete entry-section execution.

Consider a process p. During its entry section, the total iteration count of the **while** loops at statements 9b and 11b is bounded by the number of statement executions that establish $S[p] = true$. This can happen only if some other process q executes statement 8g or 15g while $q.rival = p$ holds. The arbitration-tree structure implies that this can happen only if

- process p always enters node n from side s, where $s = 0$ or 1, and

- process q executes either statement 8g or 15g while $q.node = n \land q.side = 1 - s \land q.rival = p$ holds.

Since there are $\Theta(\log N)$ nodes along the path taken by process p to reach its critical section, it suffices to prove the following lemma.

Lemma 3.3 Consider a process p and a node n. Assume that p always enters node n from side s ($s = 0$ or 1) during its entry section. During an interval in which p is in its entry section, there can be at most seven[1] events e such that e is an execution of statement 8g or 15g by some process q with $q.node = n \land q.side = 1-s \land q.rival = p$.

Proof: We represent process p's execution of statement z by $z.p[n, s]$, where n and s are the values of $p.node$ and $p.side$, respectively, before statement z is executed. We consider three cases, depending on the program counter of process p.

Case 1. Process p has not yet executed $2.p[n, s]$.

By the program text, $C[n][s] \neq p$ holds before statement $2.p[n, s]$ is executed. A process q, other than p, can establish $S[p] = true$ by executing either 8g.$q[n, 1-s]$ or 15g.$q[n, 1-s]$ *only once.* Note that, after that single event, and while Case 1 continues to hold, any process r (r could be q or another arbitrary process) executing with $r.node = n \land r.side = 1-s$ will find $r.rival \neq p$ at statement 5 or 14, and hence cannot establish $S[p] = true$.

Case 2. Process p has executed $2.p[n, s]$ but not $3.p[n, s]$ (*i.e.*, $p@\{3\} \land p.node = n \land p.side = s$ holds).
Case 3. Process p has executed $3.p[n, s]$ but not $4.p[n, s]$ (*i.e.*, $p@\{4\} \land p.node = n \land p.side = s$ holds).
Case 4. Process p has executed $4.p[n, s]$.

While each of these cases holds, a process q, other than p, can establish $S[p] = true$ *twice* by executing statements 8g.$q[n, 1-s]$ and 15g.$q[n, 1-s]$.

Assume that after q exits node n, a process r (r could be q again, or another arbitrary process) enters node n from side $1-s$. Note that statement 15g.$q[n, 1-s]$ has already established $P[n][s] \neq 0$, which is not falsified by any statement while one of Cases 2, 3, or 4 continues to hold. (By the correctness of the arbitration-tree mechanism, the only

[1]The number of events can be actually reduced to *five* with careful bookkeeping, but since this does not change the asymptotic argument, we will content ourselves with a less tight bound here.

statement that may falsify $P[n][s] \neq 0$ is $4.p[n, s]$, which falsifies Case 3 and establishes Case 4.) Therefore, process r will find $P[n][s] \neq 0$ at statement 7 if any of Cases 2, 3, or 4 continues to hold, and will not execute statements 8e and 8g.

Similarly, note that process r itself establishes $T[n] = r$ by executing $3.r[n, 1-s]$, which is not falsified by any statement while one of Cases 2, 3, or 4 continues to hold. (By the correctness of the arbitration-tree mechanism, the only statement that may falsify $T[n] = r$ is $3.p[n, s]$, which falsifies Case 2 and establishes Case 3.) Therefore, process r will find $T[n] = r$ at statement 14 if any of Cases 2, 3, or 4 continues to hold, and will not execute statements 15e and 15g.

From these arguments, it follows that $S[p] = \mathit{true}$ can be established *at most twice* while *each* of Cases 2, 3, and 4 continues to hold. Hence, we have established the following bound on the number of events that may establish $S[p] = \mathit{true}$: 1 {Case 1} + 2 {Case 2} + 2 {Case 3} + 2 {Case 4} = 7. □

Finally, from Theorem 3.1 and Lemma 3.3, we have the following.

Theorem 3.2 Algorithm F *is a correct, starvation-free mutual exclusion algorithm, with* $\Theta(N)$ *space complexity and* $\Theta(\log N)$ *RMR time complexity, on both CC and DSM systems.* □

3.3 Concluding Remarks

We have presented a mutual exclusion algorithm with $\Theta(N)$ space complexity and $\Theta(\log N)$ RMR time complexity on both CC and DSM systems. Our algorithm was created by applying a series of simple transformations to Yang and Anderson's mutual exclusion algorithm. The transformation used to obtain Algorithm LS may actually be of independent interest, because it can be applied to convert any algorithm in which spin variables are dynamically shared into one in which each process has a unique spin location. The resulting algorithm will be a local-spin algorithm on a DSM machine without coherent caches as long as the **while** loops introduced in the transformation cannot iterate unboundedly. (In Chapter 8, we introduce another transformation method for obtaining DSM algorithms, which is more general but has greater overhead.)

As described in Section 2.4, Attiya and Bortnikov presented an adaptive non-local-spin mutual exclusion algorithm under read/write atomicity [24]. Their algorithm achieves $O(M^2)$ space complexity, where M is an *a priori* upper bound on the number of concurrently active processes. Therefore, it is possible to construct an algorithm with space complexity independent of N, if one does not insist on local spinning. However, among local-spin algorithms, our algorithm is clearly optimal, because *every* process must have at least one spin variable.

In Chapter 4, we show that, by using Algorithm F as a subroutine, it is possible to construct an adaptive mutual exclusion algorithm with $\Theta(N)$ space complexity and $\Theta(\min(k, \log N))$ time complexity, where k is point contention.

CHAPTER 4

Adaptive Mutual Exclusion Under Read/Write Atomicity*

In this chapter, we present an adaptive algorithm for N-process mutual exclusion under read/write atomicity in which all busy waiting is by local spinning. In our algorithm, each process p performs $O(k)$ remote memory references to enter and exit its critical section, where k is the maximum point contention experienced by p. The space complexity of our algorithm is $\Theta(N)$, which is clearly optimal. Our algorithm is the first mutual exclusion algorithm under read/write atomicity that is adaptive under the RMR (remote-memory-reference) measure.

As described in Section 2.4, several read/write mutual exclusion algorithms have been presented that are adaptive under various time complexity measures. (As before, we let k and k' denote point and interval contention, respectively.) One of the first such algorithms was an algorithm of Styer that has $O(\min(N, k' \log N))$ step time complexity and $O(\min(N, k' \log N))$ system response time [78]. Choy and Singh presented an algorithm with $O(N)$ step time complexity and $O(k')$ system response time [30]. More recently, Attiya and Bortnikov presented an algorithm with $O(k)$ step time complexity and $O(\log k)$ system response time [24]. This algorithm was obtained by improving some of the mechanisms used in Choy and Singh's algorithm. In other work, Afek, Stupp, and Touitou [5] constructed an adaptive bakery algorithm, by combining Lamport's bakery algorithm [51] with wait-free objects. Their algorithm has $O(k^4)$ step time complexity and $O(k^4)$ system response time.

*The results presented in this chapter have been published in the following paper.
[15] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43. Lecture Notes in Computer Science 1914, Springer-Verlag, October 2000.

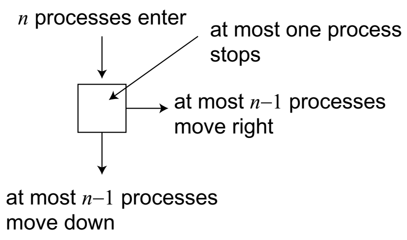
Algorithm	System response time	Step time complexity	RMR/DSM time complexity	Space complexity
Styer [78]	$O(\min(N, k' \log N))$	$O(\min(N, k' \log N))$	∞	$\Theta(N)$
Choy & Singh [30]	$O(k')$	$O(N)$	∞	$\Theta(N)$
Attiya & Bortnikov [24]	$O(\log k)$	$O(k)$	∞	$\Theta(N \log M)$
Attiya & Bortnikov [24]	$O(\log k')$	$O(k')$	∞	$\Theta(M \log M)$
Afek, *et al.* [5]	$O(k^4)$	$O(k^4)$	∞	$\Theta(N^3 + M^3 N)$
Afek, *et al.* [6]	$O((k')^2)$	$O(\min((k')^2, k' \log N))$	$O(\min((k')^2, k' \log N))$	$\Theta(N^2)$
This chapter				
Algorithm A-U	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	∞
Algorithm A-B	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$\Theta(N^2)$
Algorithm A-LS	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$\Theta(N)$

Table 4.1: Comparison of known adaptive algorithms. (This table is an extension of Table 2.1.) In this table, k denotes point contention, k' denotes interval contention, and M denotes an upper bound on the maximum number of processes concurrently active in the system (possibly less than N). (Although [5] uses a bounded number of variables, some of these variables are unbounded.)

None of the previously-cited adaptive algorithms is a local-spin algorithm, and thus each has unbounded RMR time complexity, unless write-update caches are assumed.[1] Surprisingly, while adaptivity and local spinning have been the predominant themes in recent work on mutual exclusion, the problem of designing an adaptive, local-spin algorithm under read/write atomicity has remained open until recently. In this chapter, we close this problem. In addition, Afek, Stupp, and Touitou [6] have independently devised another local-spin adaptive algorithm, based on a long-lived implementation of a splitter element (see Section 4.1.1), with a structure that is similar to our algorithm. The adaptive algorithms mentioned so far are summarized in Table 4.1.

Our algorithm can be seen as an extension of the fast-path algorithm of Anderson and Kim [14], mentioned in Section 2.3.2. That algorithm was devised by thinking about connections between fast-path mechanisms and long-lived renaming [67]. Long-lived renaming algorithms are used to "shrink" the size of the name space from which process identifiers are taken. The problem is to design operations that processes may invoke in order to acquire new names from the reduced name space when they are needed, and to release any previously-acquired name when it is no longer needed. In Anderson and Kim's algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. Our adaptive algorithm can be seen as a generalization of Anderson and Kim's fast-path mechanism

[1]See the remark in Table 2.1, and the discussion of time-complexity measures on page 31.



n processes enter
at most one process stops
at most $n-1$ processes move right
at most $n-1$ processes move down

```
shared variable  $X$: $\{\bot\} \cup \{0..N-1\}$ initially $\bot$;
                 $Y$: boolean initially $true$

private variable $dir$: $\{stop, right, down\}$

1: $X := p$;
2: if $\neg Y$ then $dir := right$
   else
3:     $Y := false$;
4:     if $X = p$ then $dir := stop$
                  else $dir := down$
       fi
   fi
```

Figure 4.1: The splitter element and the code fragment that implements it.

in which *every* name is associated with some "path" to the critical section. The length of the path taken by a process is determined by the point contention that it experiences.

An informal description of our adaptive algorithm is given in the following section. A formal correctness proof for the algorithm is given in Appendix A.

4.1 Adaptive Algorithm

In our adaptive algorithm, code sequences from several other algorithms are used. In Section 4.1.1, we present a review of these other algorithms and discuss some of the basic ideas underlying our algorithm. Then, in Sections 4.1.2–4.1.4, we present a detailed description of our algorithm.

4.1.1 Related Algorithms and Key Ideas

At the heart of our algorithm is the splitter element from the grid-based long-lived renaming algorithm of Moir and Anderson [67]. The splitter element [56], which has been described in Section 2.3.1, is shown here again in Figure 4.1. As described before, each process that invokes the splitter code either stops, moves down, or moves right (the move is defined by the value assigned to the variable *dir*). The splitter has the following key properties: if n processes invoke a splitter, then at most one of them can stop at that splitter, at most $n-1$ can move right, and at most $n-1$ can move down.

Because of these properties, it is possible to solve the renaming problem by interconnecting a collection of splitters in a grid as shown in Figure 4.2(a). (The diagonal

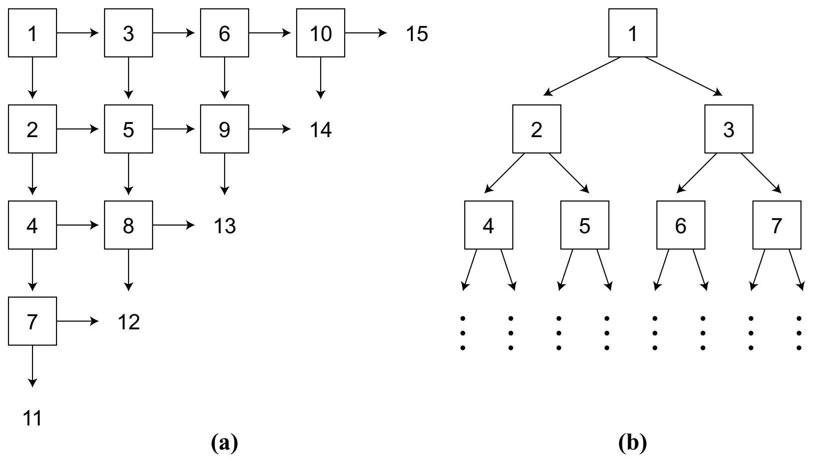


Figure 4.2: **(a)** Renaming grid (depicted for $N = 5$). **(b)** Renaming tree.

numbering scheme depicted here is due to [25].) A name is associated with each splitter. If the grid has N rows and N columns, then by induction, every process eventually stops at some splitter. When a process stops at a splitter, it acquires the name associated with that splitter. In the *long-lived* renaming problem [67], processes must have the ability to release the names they acquire. In the grid algorithm, a process can release its name by resetting (*i.e.*, reopening) each splitter on the path traversed by it in acquiring its name. (A splitter is *open* if it can be acquired once again by some process, and *closed* otherwise.) A splitter can be reset by resetting its Y variable to *true*. For the renaming mechanism to work correctly, it is important that a splitter be reset *only* if there are no processes "downstream" from it (*i.e.*, in the sub-grid "rooted" at that splitter). In Moir and Anderson's algorithm, it takes $O(N)$ time to determine if there are "downstream" processes. This is because each process checks every other process individually to determine if it is downstream from a splitter. As we shall see, a more efficient reset mechanism is needed for our adaptive algorithm.

The main idea behind our algorithm is to let an arbitration tree form dynamically within a structure similar to the renaming grid. This tree may not remain balanced, but its height is proportional to contention. The job of integrating the renaming aspects of the algorithm with the arbitration tree is greatly simplified if we replace the grid by a binary tree of splitters as shown in Figure 4.2(b). (Since we are now working with a tree, we will henceforth refer to the directions associated with a splitter as *stop*, *left*, and *right*.) Note that this results in many more names than before. However, this is not a major concern, because we are really not interested in minimizing the name space. The arbitration tree is defined by associating a three-process mutual exclusion algorithm with each node in the renaming tree. This three-process algorithm can be implemented in constant time using the local-spin mutual exclusion algorithm (Algorithm YA-2) of Yang and Anderson [84], which is described in Section 2.2.2. This algorithm may be invoked at a node by the process that stopped at that node, and one process from each of the left and right subtrees beneath that node. This is why a three-process algorithm is needed.

In our algorithm, a process p performs the following basic steps. (For the moment, we are ignoring certain complexities that must be dealt with.)

Step 1 p first acquires a new name by moving down from the root of the renaming tree, until it stops at some node. In the steps that follow, we refer to this node as p's *acquired node*. p's acquired node determines its starting point in the arbitration tree.

Step 2 p then competes within the arbitration tree by executing each of the three-process entry sections on the path from its acquired node to the root.

Step 3 After competing within the arbitration tree, p executes its critical section.

Step 4 Upon completing its critical section, p releases its acquired name by reopening all of the splitters on the path from its acquired node to the root.

Step 5 After releasing its name, p executes each of the three-process exit sections on the path from the root to its acquired node.

If we were to use a binary tree of height N, just as we previously had a grid with N row and N columns, then the total number of nodes in the tree would be $\Theta(2^N)$. We circumvent this problem by defining the tree's height to be $\log N$, which results in a tree with $\Theta(N)$ nodes. (For simplicity, we assume that N is a power of two.) With this change, a process could "fall off" the end of the tree without acquiring a name. However, this can happen only if contention is $\Omega(\log N)$. To handle processes that "fall off the end," we introduce a second arbitration tree, which is implemented using the algorithm presented in Chapter 3. We refer to the two trees used in our algorithm as the *renaming tree* and *overflow tree*, respectively. These two trees are connected by placing Algorithm YA-2, as illustrated in Figure 4.3(a). Figure 4.3(b) illustrates the steps that might be taken by a process p in acquiring a new name if contention is $O(\log N)$. Figure 4.3(c) illustrates the steps that might be taken by a process q if contention is $\Omega(\log N)$.

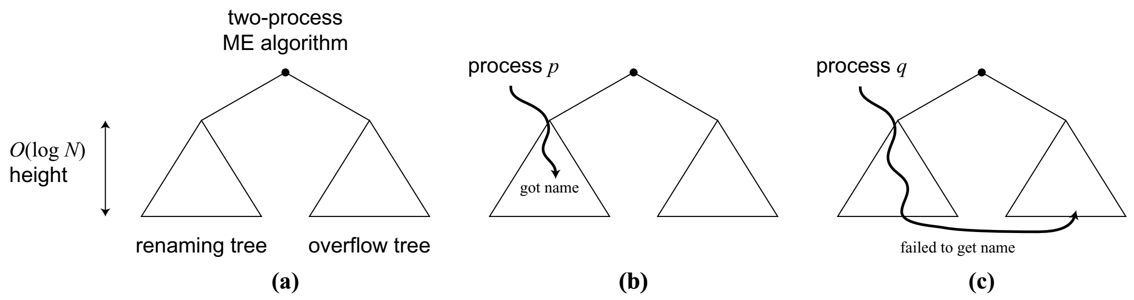


Figure 4.3: **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

A major difficulty that we have ignored until this point is that of efficiently reopening a splitter, as described in Step 4 above. In Moir and Anderson's renaming algorithm, it takes $O(N)$ time to reopen a splitter. To see why reopening a splitter is difficult, consider again Figure 4.1. If a process does succeed in stopping at a splitter, then that process can reopen the splitter itself by simply assigning $Y := \mathit{true}$. On the other hand, if no process succeeds in stopping at a splitter, then some process that moved left or right from that splitter must reset it. Unfortunately, because processes are asynchronous and communicate only by means of atomic read and write operations, it can be difficult for a left- or right-moving process to know whether some process has stopped at a splitter.

Anderson and Kim [14] solved this problem in their fast-path mutual exclusion algorithm by exploiting the fact that much of the reset code can be executed within a process's critical section. Thus, the job of designing efficient reset code is much easier here than when designing a *wait-free* long-lived renaming algorithm. As mentioned earlier, in Anderson and Kim's fast-path algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. In our adaptive algorithm, we must efficiently manage acquisitions and releases for a set of names.

Having introduced the major ideas that underlie our algorithm, we now present a detailed description of the algorithm and its properties. We do this in three steps. In Section 4.1.2, we consider a version of the algorithm in which unbounded memory is used to reset splitters in constant time. Then, in Section 4.1.3, we consider a variant of the algorithm with $\Theta(N^2)$ space complexity in which all variables are bounded. Finally, in Section 4.1.4, we present another variant that has $\Theta(N)$ space complexity. In explaining these algorithms, we actually present proof sketches for some of the key

```
const
    $L = \log N$;               /* depth of renaming tree = $O(\log N)$; for simplicity, we assume $N = 2^L$ */
    $T = 2N - 1$                                                    /* size of renaming tree = $O(N)$ */

type
    Ytype = record  free: boolean; rnd: $0..\infty$  end;              /* stored in one word */
    Dtype = (left, right, stop);                                      /* splitter moves */
    Ptype = record  node: $1..2T+1$; dir: Dtype  end                  /* path information */

shared variables
    X: array[1..T] of $0..N-1$;
    Y, Reset: array[1..T] of Ytype initially (true, 0);
    Round: array[1..T][$0..\infty$] of boolean initially false;
    Acquired: array[1..T] of boolean initially false

private variables
    node: $1..2T+1$;
    n: 1..T;
    level, j: $0..L+1$;
    y: Ytype;
    dir: Dtype;
    path: array[0..L] of Ptype
```

Figure 4.4: Variables used in ALGORITHM A-U.

properties of each algorithm. Our intent is to use these proof sketches as a means for intuitively explaining the basic mechanisms of each algorithm. A formal correctness proof for the final algorithm is presented in Appendix A.

4.1.2 ALGORITHM A-U: Unbounded Algorithm

The first algorithm, which we call ALGORITHM A-U (for adaptive and unbounded), is shown in Figure 4.5. Variables used in this algorithm are shown in Figure 4.4. Before describing how this algorithm works, we first examine its basic structure. At the top of Figure 4.4, definitions of two constants are given: L, which is the maximum level in the renaming tree (the root is at level 0), and T, which gives the total number of nodes in the renaming tree. As mentioned earlier, the renaming tree is comprised of a collection of splitters. These splitters are indexed from 1 to T. If splitter i is not a leaf, then its left and right children are splitters $2i$ and $2i + 1$, respectively.

Each splitter i is accessed by three subroutines *AcquireNode*(i), *ReleaseNode*(i, dir), and *ClearNode*(i). Function *AcquireNode*(i) determines if the process can acquire splitter i by executing a modified version of the splitter code in Figure 4.1, and returns *stop*, *left*, or *right* depending on its outcome. A process invokes *ReleaseNode*(i, dir) in

```
process $p$ ::    /* $0 \le p < N$ */
while $true$ do
0:  Noncritical Section;
1:  $node,\ level := 1,\ 0$;
    /* descend renaming tree */
    repeat
2:      $dir := AcquireNode(node)$;
3:      $path[level] := (node,\ dir)$;
        if $dir = left$ then
            $level := level + 1$;
            $node := 2 \cdot node$
        elseif $dir = right$ then
            $level := level + 1$;
            $node := 2 \cdot node + 1$
        fi
    until $(level > L) \vee (dir = stop)$;
    if $level \le L$ then      /* got a name */
        /* compete in the renaming tree,
                and then 2-process algorithm */
        for $j := level$ downto 0 do
4:          $\mathtt{Entry}_3(path[j].node,\ path[j].dir)$
        od;
5:      $\mathtt{Entry}_2(0)$
    else      /* did not get a name */
        /* compete in the overflow tree,
                and then 2-process algorithm */
6:      $\mathtt{Entry}_N(p)$;
7:      $\mathtt{Entry}_2(1)$
    fi;
8:  Critical Section;
    /* reset splitters */
    for $j := \min(level,\ L)$ downto 0 do
        $n,\ dir := path[j].node,\ path[j].dir$;
9:      $ReleaseNode(n, dir)$
    od;
    /* execute appropriate exit sections */
    if $level \le L$ then
10:     $\mathtt{Exit}_2(0)$;
        for $j := 0$ to $level$ do
11:         $\mathtt{Exit}_3(path[j].node,\ path[j].dir)$ od;
12:     $ClearNode(node)$
    else
13:     $\mathtt{Exit}_2(1)$;
14:     $\mathtt{Exit}_N(p)$
    fi
od
```

```
function $AcquireNode(n\colon 1..T)\colon Dtype$
15: $X[n] := p$;
16: $y := Y[n]$;
    if $\neg y.free$ then return $right$ fi;
17: $Y[n] := (false,\ 0)$;
18: if $X[n] \ne p\ \vee$
19:        $Acquired[n]$ then
        return $left$
    fi;
20: $Round[n][y.rnd] := true$;
21: if $Reset[n] \ne y$ then
22:     $Round[n][y.rnd] := false$;
        return $left$
    fi;
23: $Acquired[n] := true$;
    return $stop$

procedure $ReleaseNode(n\colon 1..T,\ dir\colon Dtype)$
24: if $dir = right$ then return fi;
25: $y := Reset[n]$;
26: $Reset[n] := (false,\ y.rnd)$;
27: if $dir = stop\ \vee\ \neg Round[n][y.rnd]$ then
28:     $Reset[n] := (true,\ y.rnd + 1)$;
29:     $Y[n] := (true,\ y.rnd + 1)$
    fi

procedure $ClearNode(n\colon 1..T)$
30: $Acquired[n] := false$
```

Figure 4.5: ALGORITHM A-U: Adaptive algorithm with unbounded memory.

its exit section (statement 9) if and only if it has invoked *AcquireNode*(i) in its entry section and obtained a return value of *dir*. Similarly, a process invokes *ClearNode*(i) in its exit section (statement 12) if and only if it has stopped at splitter i in its entry section (*i.e.*, its call to *AcquireNode*(i) returned *stop*). These two procedures "reopen" splitter i for future use, as explained in detail later.

Each splitter i is defined by four shared variables and an infinite shared array: $X[i]$, $Y[i]$, *Reset*$[i]$, *Round*$[i]$ (the array), and *Acquired*$[i]$. Variables $X[i]$ and $Y[i]$ are as in Figure 4.1, with the exception that $Y[i]$ now has an additional integer *rnd* field. As explained below, Algorithm A-U works by associating "round numbers" with the various rounds of competition for the name corresponding to each splitter. In Algorithm A-U, these round numbers grow without bound. The *rnd* field of $Y[i]$ gives the current round number for splitter i. *Reset*$[i]$ is used to reinitialize the *rnd* field of $Y[i]$ when name i is released. *Round*$[i][r]$ is used to identify a potential "winning" process that has succeeded in acquiring name i in round r. *Acquired*$[i]$ is set when some process acquires name i.

Each process descends the renaming tree, starting at the root, until it either acquires a name or "falls off the end" of the tree, as discussed earlier. A process determines if it can acquire name i by invoking *AcquireNode*(i) at statement 2. Statement 3 simply prepares for the next iteration of the **repeat** loop (if there is one). In *AcquireNode*(i), statements 15–18 correspond to the splitter code in Figure 4.1, and statements 19–23 are executed as part of a handshaking mechanism that prevents a process that is releasing a name from adversely interfering with processes attempting to acquire that name; this mechanism is discussed in detail below.

If a process p succeeds in acquiring a name while descending within the renaming tree, then it competes within the renaming tree by moving up from its acquired name to the root, executing the three-process entry sections on this path (statement 4). Each of these three-process entry sections is denoted "$\texttt{Entry}_3(n, d)$," where n is the corresponding tree node, and d is the "identity" of the invoking process. The "identity" that is used is simply the invoking process's direction out of node n (*stop*, *left*, or *right*) when it descended the renaming tree. After ascending the renaming tree, p invokes the two-process entry section "on top" of the renaming and overflow trees (as illustrated in Figure 4.3(a)) using "0" as a process identifier (statement 5). This entry section is denoted "$\texttt{Entry}_2(0)$."

If a process p does *not* succeed in acquiring a name while descending within the renaming tree, then it competes within the overflow tree (statement 6), which is implemented using Yang and Anderson's N-process arbitration-tree algorithm. The entry section of this algorithm is denoted $\texttt{Entry}_N(p)$. Note that p uses its own process identifier in this algorithm. After competing within the overflow tree, p executes the two-process algorithm "on top" of both trees using "1" as a process identifier (statement 7). This entry section is denoted "$\texttt{Entry}_2(1)$."

After completing the appropriate two-process entry section, process p executes its critical section (statement 8). It then resets each of the splitters that it visited while descending the renaming tree by invoking *ReleaseNode* at statement 9. This reset mechanism is discussed in detail below. Process p then executes the exit sections corresponding to the entry sections it executed previously (statements 10–14). The exit sections are specified in a manner that is similar to the entry sections. (Statement 12 is discussed below.)

We now consider in detail the three subroutines that are executed to acquire or reset some splitter i. To facilitate this discussion, we will index these statements by i. For example, when we refer to the execution of statement $17[i]$ by process p, we mean the execution of statement 17 by p when its argument n equals i. Throughout this discussion, we use the following notational conventions.

As in Chapter 3, we use $s.p$ to denote the statement with label s of process p, $p.v$ to represent p's private variable v, and $p@S$ to denote that the program counter for process p equals some value in the set S. Also, as before, we assume that each labeled sequence of statements is atomic. For example, consider statement $18.p$. If $18.p$ is executed while $X[p.n] = p$ holds, then it establishes $p@\{19\}$. On the other hand, if $18.p$ is executed while $X[p.n] \neq p$ holds, then it returns *left* from *AcquireNode* and establishes $p@\{3\}$. Statements 2, 9, and 12 are considered as *branches* to each subroutine, and do nothing but establishing $p@\{15, 24, 30\}$, respectively.

We number statements in this manner to reduce the number of cases that must be considered in the proof. Note that each numbered sequence of statements reads or writes at most one shared variable. (Because the `ENTRY` and `EXIT` routines are assumed to be correct, we can assume that they execute atomically and do not access any of the shared variables of ALGORITHM A-U.)

As explained above, one of the problems with the splitter code is that it is difficult for a left- or right-moving process at splitter i to know which (if any) process has acquired name i. In ALGORITHM A-U, this problem is solved by viewing the computation involving each splitter as occurring in a sequence of rounds. Each round ends when the splitter is reset. During a round, at most one process succeeds in acquiring the name of the splitter. Note that it is possible that *no* process acquires the name during a round. So that processes can know the current round number at splitter i, an additional *rnd* field has been added to $Y[i]$. In essence, the round number at splitter i is used as a temporary identifier to communicate with the winning process (if any) at splitter i. This identifier will increase without bound over time, so the potential winner of each round can be uniquely identified. Formally, we have the following definitions.

Definition An *execution interval* is a set of consecutive states in a particular execution history. It is denoted (e, f), where e is either 0 (indicating that the interval starts with the initial state) or some event, and f is either ∞ (indicating that the interval never terminates) or another event that is executed after e. Interval (e, f) contains all states after the execution of e and before the execution of f. We say that (e, f) is *infinite* if $f = \infty$, and *finite* otherwise. $\Box$

Definition A *round* of a splitter i is an execution interval $H = (e, f)$ satisfying the following.

- e is either 0 or an event that writes $Y[i] := (\mathit{true}, r)$, for some r.
- f is either ∞ or another event that writes $Y[i] := (\mathit{true}, s)$, for some s.
- For any event g inside H (excluding e and f), if g writes $Y[i]$, then g writes $Y[i] := (\mathit{false}, 0)$.

We say that r is the *round number* of H. (If $e = 0$, then the round number of H is 0, the initial value of $Y[i].\mathit{rnd}$.) We also use $\mathcal{R}(i, r)$ to denote round r of splitter i. $\Box$

In order to show the correctness of Algorithm A-U, we need several properties, given below. The following property follows immediately from the correctness of the original splitter code.

Property 1: Let $\mathcal{S}$ be the set of processes that execute statement 16[i] during round $\mathcal{R}(i, r)$. Then, at most one process p in $\mathcal{S}$ reaches statement 20[i]. $\Box$

We say that p is the *winner* of round $\mathcal{R}(i, r)$ if p executes statement 16[i] during $\mathcal{R}(i, r)$ and then stops at splitter i (*i.e.*, returns *stop* at statement 23[i]). By Property 1, if a winner exists, then it is uniquely defined.

The following property asserts that *Reset*[i] correctly indicates the current round number.

Property 2: During round $\mathcal{R}(i, r)$, either $\mathit{Reset}[i].\mathit{rnd} = r$ holds, or there exists a process p satisfying $p@\{29\} \wedge p.n = i$ (*i.e.*, p is about to execute statement $29[i]$). Moreover, in the latter case, $\mathit{Reset}[i].\mathit{rnd} = r + 1$ holds.

Proof: Note that $\mathit{Reset}[i]$ is updated only inside $\mathit{ReleaseNode}(i)$. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, invocations of *ReleaseNode* (by different processes) cannot overlap. From this, Property 2 easily follows. $\Box$

With the added *rnd* field, a left- or right-moving process at splitter i has a way of identifying a process that has acquired the name at splitter i. To see how this works, consider what happens during round $\mathcal{R}(i, r)$. By Property 1, of the processes that execute statement $16[i]$ during $\mathcal{R}(i, r)$, at most one will reach statement $20[i]$. A process that reaches statement $20[i]$ will either stop at node i by executing statement $23[i]$ (*i.e.*, become the winner of $\mathcal{R}(i, r)$) or be deflected left. This gives us two cases to analyze, depending on whether $\mathcal{R}(i, r)$ has a winner or not. These two cases are considered in the following two properties.

Property 3: Assume that $\mathcal{R}(i, r)$ has a winner p. Then, $\mathcal{R}(i, r)$ does not end until p executes statement $29[i]$.

Proof: By definition, $\mathcal{R}(i, r)$ may end only if some process q executes statement $29[i]$. For the sake of contradiction, assume that some other process $q \neq p$ executes statement $29[i]$ during $\mathcal{R}(i, r)$ (which would then terminate $\mathcal{R}(i, r)$). Without loss of generality, assume that $\mathcal{R}(i, r)$ is the first round of splitter i in which this happens.

Consider the state at which q executes statement $29[i]$. Because p has executed statement $16[i]$ but has not yet executed statement $29[i]$, $p@\{17[i]..23[i], 3..5, 8, 9, 24[i]..29[i]\}$ holds at that state. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, q cannot execute statement $29[i]$ while $p@\{8, 9, 24[i]..29[i]\}$ holds. Also note that statement $29[i].q$ starts a new round, say, $\mathcal{R}(i, s')$, where $s' > r$. Thus, if p executes statement $21[i]$ *after* q executes statement $29[i]$, then $21[i].p$ is executed during some round $\mathcal{R}(i, s)$, where $s \geq s' > r$. Therefore, by Property 2, p will find either $\mathit{Reset}[i].\mathit{rnd} = s$ or $\mathit{Reset}[i].\mathit{rnd} = s + 1$ when it executes statement $21[i]$. In either case, we have $\mathit{Reset}[i].\mathit{rnd} > r$, and hence p cannot stop at splitter i.

The only remaining possibility is that $p@\{23[i], 3..5\}$ holds when q executes statement $29[i]$. (Note that, in this case, if q *were* to reopen splitter i then we could end up with two processes concurrently invoking $\texttt{Entry}_3(i, \mathit{stop})$ and $\texttt{Exit}_3(i, \mathit{stop})$ at statements 4 and 11, *i.e.*, both processes use i as a "process identifier." The `Entry` and `Exit` calls obviously cannot be assumed to work correctly if such a scenario could happen.)

So, assume that q executes statement 29[i] while $p@\{23[i], 3..5\}$ holds. Note that the round of splitter i can change only by some process executing statement 29[i] inside $\mathit{ReleaseNode}(i)$. Since invocations of *ReleaseNode* (by different processes) cannot overlap, it follows that q executes statements 25[i]–29[i] during $\mathcal{R}(i, r)$. Thus, by Property 2, q reads $\mathit{Reset}[i].\mathit{rnd} = r$ at statement 25[i].

Thus, for q to execute statement 29[i], it must find $q.\mathit{dir} = \mathit{stop} \vee \mathit{Round}[i][r] = \mathit{false}$ at statement 27[i]. First, if $q.\mathit{dir} = \mathit{stop}$ holds, then q has stopped at splitter i, *i.e.*, q is the winner of round $\mathcal{R}(i, s)$, for some s. Combined with our assumption that q executes statement 29[i] during round $\mathcal{R}(i, r)$, it follows that round $\mathcal{R}(i, s)$ has ended (and round $\mathcal{R}(i, r)$ has started) before q executes statement 29[i], which contradicts our assumption that $\mathcal{R}(i, r)$ is the first such round.

It follows that q reads $\mathit{Round}[i][r] = \mathit{false}$ at statement 27[i]. For this to happen, q must execute statement 27[i] before p assigns $\mathit{Round}[i][r] := \mathit{true}$ at statement 20[i]. Hence, statement 26[i] is executed by q before statement 21[i] is executed by p. Thus, p must find $\mathit{Reset}[i] \neq p.y$ at statement 21[i], *i.e.*, it is deflected left at splitter i, a contradiction. $\Box$

The following property pertains to the case when $\mathcal{R}(i, r)$ has no winner — in this case, some process that is deflected left eventually reopens splitter i.

Property 4: Assume that round $\mathcal{R}(i, r)$ has no winner. If some process executes statement 17[i] during $\mathcal{R}(i, r)$, then some process eventually executes statement 29[i].

Proof: Let $\mathcal{S}$ be the (nonempty) set of processes that execute statement 17[i] during round $\mathcal{R}(i, r)$. We claim that at least one process p in $\mathcal{S}$ finds $\mathit{Round}[i][r]$ to be false at statement 27[i]. We consider two cases.

First, assume that there exists some process p that executes statement 16[i] during $\mathcal{R}(i, r)$, and subsequently executes statement 20[i]. By Property 1, p is unique. Since $\mathcal{R}(i, r)$ has no winner, p must be deflected left by executing statements 21[i] and 22[i], re-establishing $\mathit{Round}[i][r] = \mathit{false}$. Thus, p eventually reads $\mathit{Round}[i][r] = \mathit{false}$ at statement 27[i] (unless some other process has already done so and executed statement 29[i]). Note that, because the `ENTRY` and `EXIT` routines are assumed to be correct, and because Algorithm A-U does not contain any busy-waiting loops elsewhere, p cannot be stalled indefinitely without executing statement 27[i].

Second, assume that there exists no process that executes statement 16[i] during $\mathcal{R}(i, r)$, and then executes statement 20[i]. In this case, since *Round*[i][r] is initially *false*, it remains *false* throughout the execution of the algorithm. Thus, some process eventually reads *Round*[i][r] = *false* at statement 27[i]. □

By Property 1 and Property 3, if a process p stops at splitter i, then no other process stops at splitter i, and the splitter remains closed until p finishes execution of *ReleaseNode*(i). Note that the assignments to *Acquired*[i] at statements 23[i] and 30[i] prevent the reopening of splitter i from actually taking effect until after p has finished executing its exit section. This is the reason we need two separate procedures *ReleaseNode* and *ClearNode*, so that the former is executed effectively within critical sections and the latter, outside of critical sections.

Note that splitter i is closed if and only if some process establishes $Y[i] = (\mathit{false}, 0)$ by executing statement 17[i]. Thus, by Property 4, if no process stops at splitter i, and if splitter i becomes closed, then it is eventually reopened.

Because the splitters are always reset properly, it follows that the `ENTRY` and `EXIT` routines are always invoked properly. If these routines are implemented using Algorithm YA-2, then since that algorithm is starvation-free, Algorithm A-U is as well.

Having dispensed with basic correctness, we now informally argue that Algorithm A-U is adaptive under the RMR measure. (For the sake of brevity, we give a rigorous proof of adaptive only for the final algorithm (Algorithm A-LS), given later in Section 4.1.4.) For a process p to descend to a splitter at level l in the renaming tree, it must have been deflected left or right at each prior splitter it accessed. Just as with the original grid-based long-lived renaming algorithm [67], this can only happen if the point contention experienced by p is $\Omega(l)$. Note that the time complexity per level of the renaming tree is constant. Moreover, with the `ENTRY` and `EXIT` calls implemented using Algorithm YA-2, the $\texttt{Entry}_2$, $\texttt{Exit}_2$, $\texttt{Entry}_3$, and $\texttt{Exit}_3$ calls take constant time, and the $\texttt{Entry}_N$ and $\texttt{Exit}_N$ calls take $\Theta(\log N)$ time. Note that the $\texttt{Entry}_N$ and $\texttt{Exit}_N$ routines are called by a process only if its point contention is $\Omega(\log N)$. Thus, we have the following.

Lemma 4.1 Algorithm A-U is a correct, starvation-free mutual exclusion algorithm with $O(\min(k, \log N))$ RMR time complexity and unbounded space complexity. □

```
/∗ all constant, type, and private variable declarations are as defined in Figure 4.5
   except as noted here ∗/

type
   $Ytype$ = record  $free$: boolean; $rnd$: $0..N-1$  end /∗ stored in one word ∗/

shared variables
   $X$: array[$1..T$] of $0..N-1$;
   $Y$, $Reset$: array[$1..T$] of $Ytype$ initially ($true$, 0);
   $Round$: array[$1..T$][$0..N-1$] of boolean initially $false$;
   $Obstacle$: array[$0..N-1$] of $0..T$ initially 0;
   $Acquired$: array[$1..T$] of boolean initially $false$

process $p$ ::   /∗ $0 \le p < N$ ∗/
function $AcquireNode$($n$: $1..T$): $Dtype$
15: $X[n] := p$;
16: $y := Y[n]$;
    if $\neg y.free$ then return $right$ fi;
17: $Y[n] := (false,\ 0)$;
18: $Obstacle[p] := n$;
19: if $X[n] \neq p\ \vee$
20:       $Acquired[n]$ then
        return $left$
    fi;
21: $Round[n][y.rnd] := true$;
22: if $Reset[n] \neq y$ then
23:     $Round[n][y.rnd] := false$;
        return $left$
    fi;
24: $Acquired[n] := true$;
    return $stop$
```

```
procedure $ReleaseNode$($n$: $1..T$, $dir$: $Dtype$)
25: $Obstacle[p] := 0$;
26: if $dir = right$ then return fi;
27: $Y[n] := (false,\ 0)$;
28: $X[n] := p$;
29: $y := Reset[n]$;
30: $Reset[n] := (false,\ y.rnd)$;
31: if $(dir = stop\ \vee\ \neg Round[n][y.rnd])\ \wedge$
32:       $Obstacle[y.rnd] \neq n$ then
33:     $Reset[n] := (true,\ y.rnd + 1 \bmod N)$;
34:     $Y[n] := (true,\ y.rnd + 1 \bmod N)$
    fi;
35: if $dir = stop$ then $Round[n][y.rnd] := false$ fi

procedure $ClearNode$($n$: $1..T$)
36: $Acquired[n] := false$
```

Figure 4.6: ALGORITHM A-B: Adaptive algorithm with $\Theta(N^2)$ space complexity. Statements 1–14 are identical to ALGORITHM A-U and not shown here.

Of course, the problem with ALGORITHM A-U is that the *rnd* field of $Y[i]$ that is used for assigning round numbers grows without bound. We now consider a variant of ALGORITHM A-U in which space is bounded.

4.1.3 ALGORITHM A-B: Bounded Algorithm

In ALGORITHM A-B (for "adaptive with bounded space"), $Y[i].rnd$ is incremented modulo-N, and hence does not grow without bound. ALGORITHM A-B is shown in Figure 4.6. The only new variable is *Obstacle*, which acts as an "obstacle" to prevent $Y[i].rnd$ to cycle modulo-N as long as there is a possibility of "interference," as explained below.

With $Y[i].rnd$ being incremented modulo-N, the following potential problem arises. A process p may reach statement $21[i]$ in Figure 4.6 with $y.rnd = r$ and then be delayed. While delayed, other processes may repeatedly increment $Y[i].rnd$ (statement $34[i]$) until it "cycles back" to r. Another process q could then reach statement $21[i]$ with $y.rnd = r$. This is a problem because p and q may interfere with each other in updating $Round[i][r]$.

Algorithm A-B prevents such a scenario from happening by preventing $Y[i].rnd$ from cycling while a process p that stops (or may stop) at splitter i executes within $AcquireNode(i)$. Informally, cycling is prevented by requiring process p to erect an "obstacle" that prevents $Y[i].rnd$ from being incremented beyond the value p. More precisely, note that before reaching statement $21[i]$, process p must first assign $Obstacle[p] := i$ at statement $18[i]$. Note further that before a process can increment $Y[i].rnd$ from r to $r + 1$ **mod** N (statement $34[i]$), it must first read $Obstacle[r]$ (statement $32[i]$) and find it to have a value different from i. This check prevents $Y[i].rnd$ from being incremented beyond the value p while p executes within $AcquireNode(i)$. Note that process p resets $Obstacle[p]$ to 0 at statement 25. This is done to ensure that p's own obstacle does not prevent it from incrementing a splitter's round number. The discussion so far is formalized in the following property.

Property 5: If a process p executes statement $16[i]$ during round $\mathcal{R}(i, r)$ and subsequently reaches statement $21[i]$, then $\mathcal{R}(i, p + 1 \textbf{ mod } N)$ does not start until p returns from $AcquireNode(i)$.

Proof: For the sake of contradiction, assume that a process q executes statements $25[i]$–$34[i]$ during $\mathcal{R}(i, p)$, and starts round $\mathcal{R}(i, p + 1 \textbf{ mod } N)$, while $p@\{17[i]..24[i]\}$ holds. We consider three cases.

First, if q executes statement $28[i]$ before p executes statement $15[i]$, then q establishes $Y[i] = (\mathit{false}, 0)$ at statement $27[i]$ before p executes statement $16[i]$. Moreover, because we assume the correctness of the `Entry` and `Exit` routines, no other process may reopen splitter i (by establishing $Y[i].\mathit{free} = \mathit{true}$) until q executes $34[i]$. It follows that p is deflected right at statement $16[i]$, and does not reach statement $21[i]$.

Second, assume that q executes statement $28[i]$ while $p@\{16[i]..19[i]\}$ holds. In this case, p finds $X[i] \neq p$ at statement $19[i]$ and is deflected left without reaching statement $21[i]$.

Finally, assume that q executes statement $28[i]$ *after* p executes statement $19[i]$. In this case, p has executed statement $18[i]$ and established $Obstacle[i] = p$, which continues to hold while p executes inside *AcquireNode*(i). Therefore, q reads *Obstacle*$[p] = i$ at statement $32[i]$, and does not execute statements $33[i]$ and $34[i]$. (Note that, by Property 2, q reads *Reset*$[n].rnd = p$ at statement $29[i]$.) □

Since round numbers cycle modulo-N, we have the following: if a process p executes statement $16[i]$ during round $\mathcal{R}(i, r)$ and reaches statement $21[i]$, then $Y[i].rnd = r$ is not re-established until p returns from *AcquireNode*(i). Therefore, the recycling of round numbers does not affect the correctness of ALGORITHM A-B.

The only statement not explained so far is statement 35, which simply resets the *Round* variable to be used again. From the discussion above, we have the following lemma.

Lemma 4.2 ALGORITHM A-B is a correct, starvation-free mutual exclusion algorithm with $O(\min(k, \log N))$ RMR time complexity and $\Theta(N^2)$ space complexity. □

The $\Theta(N^2)$ space complexity of ALGORITHM A-B is due to the *Round* array. We now show that this $\Theta(N^2)$ array can be replaced by a $\Theta(N)$ linked list.

4.1.4 ALGORITHM A-LS: Linear-space Algorithm

The final algorithm we present in this chapter is depicted in Figure 4.7. We refer to this algorithm as ALGORITHM A-LS (for "adaptive with linear space"). In ALGORITHM A-LS, a common pool of round numbers ranging over $\{1, \ldots, S\}$ is used for all splitters in the renaming tree. As we shall see, $O(N)$ round numbers suffice. In ALGORITHM A-B, our key requirement for round numbers was that they not be reused "prematurely." With a common pool of round numbers, a process should not choose r as the next round number for some splitter if there is a process *anywhere* in the renaming tree that "thinks" that r is the current round number of some splitter it has accessed.

Fortunately, since each process selects new round numbers within its critical section, it is fairly easy to ensure this requirement. All that is needed are a few extra data structures that track which round numbers are currently in use. These data structures replace the *Obstacle* array of ALGORITHM A-B. The main new data structure is a queue *Free* of round numbers. In addition, there is a new shared array *Inuse*, and a new shared variable *Check*. We assume that the *Free* queue can be manipulated by the following operations.

```
/* all constant, type, and private variable declarations are as defined in Figure 4.6
   except as noted here */

const
   $S = T + 2N$            /* number of possible round numbers = $O(N)$ */

type
   Ytype = record   free: boolean; rnd: $0..S$   end /* stored in one word */

shared variables
   $X$: array[$1..T$] of $0..N-1$;
   $Y$, Reset: array[$1..T$] of Ytype;
   Round: array[$1..S$] of boolean initially false;
   Free: queue of integers;
   Inuse: array[$0..N-1$] of $0..S$ initially 0;
   Check: $0..N-1$ initially 0;
   Acquired: array[$1..T$] of boolean initially false

initially                                                    private variables
   $(\forall i : 1 \leq i \leq T :: Y[i] = (true,\ i) \wedge$                ptr: $0..N-1$;
      $Reset[i] = (true,\ i)) \wedge$                               nextrnd: $1..S$;
      $(Free = (T+1) \rightarrow (T+2) \rightarrow \cdots \rightarrow S)$             usedrnd: $0..S$

process $p$ ::    /* $0 \leq p < N$ */
function AcquireNode($n$: $1..T$): Dtype
15: $X[n] := p$;
16: $y := Y[n]$;
    if $\neg y.free$ then return right fi;
17: $Y[n] := (false,\ 0)$;
18: $Inuse[p] := y.rnd$;
19: if $X[n] \neq p\ \vee$
20:        $Acquired[n]$ then
       return left
    fi;
21: $Round[y.rnd] := true$;
22: if $Reset[n] \neq y$ then
23:     $Round[y.rnd] := false$;
        return left
    fi;
24: $Acquired[n] := true$;
    return stop

procedure ReleaseNode($n$: $1..T$, dir: Dtype)
25: if $dir = right$ then return fi;
26: $Y[n] := (false,\ 0)$;
27: $X[n] := p$;
28: $y := Reset[n]$;
29: $Reset[n] := (false,\ y.rnd)$;
30: if $(dir = stop\ \vee\ \neg Round[y.rnd])$ then
31:     $ptr := Check$;
32:     $usedrnd := Inuse[ptr]$;
33:     if $usedrnd \neq 0$ then
            $MoveToTail(Free,\ usedrnd)$ fi;
34:     $Check := ptr + 1 \bmod N$;
35:     $Enqueue(Free,\ y.rnd)$;
36:     $nextrnd := Dequeue(Free)$;
37:     $Reset[n] := (true,\ nextrnd)$;
38:     $Y[n] := (true,\ nextrnd)$
    fi;
    if $dir = stop$ then
39:     $Round[y.rnd] := false$;
40:     $Inuse[p] := 0$
    fi

procedure ClearNode($n$: $1..T$)
41: $Acquired[n] := false$
```

Figure 4.7: Algorithm A-LS: adaptive algorithm with $\Theta(N)$ space complexity. Statements 1–14 are identical to Algorithm A-U and not shown here.

- *Enqueue*(*Free*, i: $1..S$): Enqueues the integer i onto the end of *Free*. (We assume that i is not already contained in *Free*.)

- *Dequeue*(*Free*): $1..S$: Dequeues the element at the head of *Free*, and returns that element.

- *MoveToTail*(*Free*, i: $1..S$): If i is in *Free*, then it is moved to the end of the queue; otherwise, do nothing.

If the *Free* queue is implemented as a doubly-linked list, then each of these operations can be performed in constant time. We stress that all of these operations are executed *only* within critical sections, *i.e.*, *Free* is really a *sequential* data structure.

The only difference beetween ALGORITHMS A-B and A-LS before the critical section is statement $18[i]$: instead of updating *Obstacle*$[p]$, process p now marks the round number r it just read from $Y[i]$ as being "in use" by assigning *Inuse*$[p] := r$. The only other differences are in *ReleaseNode*. Statements 31–34 are executed to ensure that no round number currently "in use" can propagate to the head of the *Free* queue. In particular, if a process p is delayed after having obtained r as the current round number for some splitter, then while it is delayed, r will be moved to the end of the *Free* queue by every N^{th} critical-section execution. (A similar mechanism is used in the constant-time implementation of load-linked and store-conditional from read and compare-and-swap of Anderson and Moir [21].) With $S = T + 2N$ round numbers, this is sufficient to prevent r from reaching the head of the queue while p is delayed. (Among the $S = T + 2N$ round numbers, T of them are assigned to the splitters. The *Free* queue needs $2N$ round numbers because the calls to *Dequeue* and *MoveToTail* can cause a round number to migrate toward the head of the *Free* queue by two positions per critical-section execution.)

Statement $35[i]$ enqueues the current round number for splitter i onto the *Free* queue. Statement $36[i]$ dequeues a new round number from *Free*. The mechanism explained above (statements 31–34) guarantees that the newly dequeued round number is not used anywhere in the renaming tree. The rest of the algorithm is the same as before. From the discussion so far, we have the following property.

Property 6: If a process p executes statement $16[i]$ during round $\mathcal{R}(i, r)$ and subsequently reaches statement $21[i]$, then the following holds until p returns from *AcquireNode*(i).

Either round $\mathcal{R}(i, r)$ continues, or the round number r is contained in Free. Moreover, r is not used for any other splitter.

Proof: (Formally, Property 6 is implied by invariants (I5)–(I8) and (I15), stated in Appendix A.) $\mathcal{R}(i, r)$ may end only if a process q starts a new round at splitter i by executing statement $38[i]$. In this case, q enqueues r at statement $35[i]$.

We now show that r is not used for any other splitter. For the sake of contradiction, assume that r *is* used by some other splitter. Moreover, assume that r is the first round number in which this happens.

Since each splitter initially has a distinct round number, this may happen only if some process q' dequeues r from *Free* at statement $36[h]$ (for some splitter h), and then starts $\mathcal{R}(h, r)$ at statement $38[h]$. Since *Free* has length $2N$, *ReleaseNode* must have been executed at least N times since q has enqueued r at statement $35[i]$. (Recall that invocations of *ReleaseNode* are protected by `Entry`/`Exit` calls and do not overlap.) Moreover, among the *last* N invocations of *ReleaseNode* (up to and including the invocation by q'), no execution must have invoked $\mathit{MoveToTail}(\mathit{Free}, r)$ at statement 33.

Since *Check* is incremented modulo-N, among these last N invocations of *ReleaseNode*, there exists one (by some process q'') in which statements 31 and 32 are executed while $\mathit{Check} = p$ holds.

First, assume that q'' executes statement 32 *after* p executes statement $18[i]$. In this case, q'' subsequently invokes $\mathit{MoveToTail}(\mathit{Free}, r)$ at statement 33, a contradiction. Second, if q'' executes statement 32 *before* p executes statement $18[i]$, then due to statements $26.q''$ and $27.q''$, p must either find $Y[i] = (\mathit{false}, 0)$ at statement $16[i]$, or find $X[i] \neq p$ at statement $19[i]$. (The reasoning for this is the same as in the proof of Property 5.) Thus, p cannot reach statement $21[i]$, a contradiction. $\square$

Since round numbers are never reused "prematurely," it follows that Algorithm A-LS is a correct, starvation-free mutual exclusion algorithm.

Proof of adaptivity. We now prove that Algorithm A-LS is adaptive under the RMR measure. In our proof, it is necessary to track each process's current location in the renaming tree so that we can determine when a process will be deflected left or right from some splitter. The location of a process p during its entry and exit section is depicted in Figure 4.8. In its entry section, p can only deflect other processes at the splitter it is attempting to acquire. Thus, it has a single location (Figure 4.8(a)). However, at any given instant inside its exit section, p can deflect other processes at

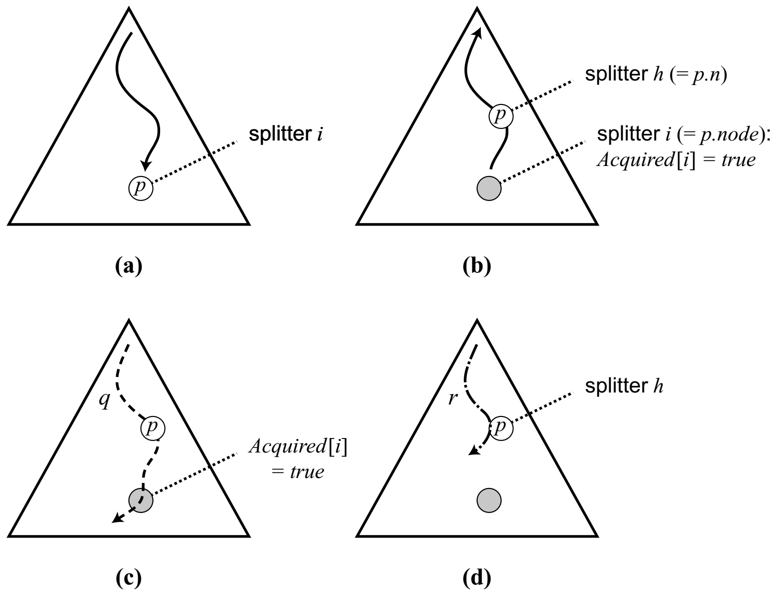


Figure 4.8: Location of a process p during its entry and exit section. **(a)** p acquires splitter i: both p and $p + N$ are located at i. **(b)** In its exit section, p resets each node h in its path by calling *ReleaseNode*(h). p is located at h, while $p + N$ is located at i. **(c)** Another process q in its entry section descends to splitter i, and is deflected left because it finds $Acquired[i] = true$ (statement 20$[i]$). **(d)** Yet another process r in its entry section descends to splitter h, and is deflected left or right because p updated $Y[h]$, $X[h]$, or $Reset[h]$ (statements 26$[h].p$, 27$[h].p$, 29$[h].p$, and 37$[h].p$).

two splitters — the splitter that it has acquired in its entry section (which is indicated by $p.node$) and the splitter it is currently resetting (which is indicated by $p.n$ inside *ReleaseNode*; see Figure 4.8(b)). In the former case, other processes may be deflected because $Acquired[p.node]$ is true (Figure 4.8(c)). In the latter case, p may deflect other processes by changing the value of Y, X, or *Reset* (statements 26–29 and 37; Figure 4.8(d)). In order to facilitate the following discussion, we associate a *shadow process* with an identifier of $p + N$ with each process p. When p is in its entry section, p and $p + N$ are always located at the same splitter indicated by $p.node$. However, when p is in its exit section, we say that p is located at the splitter indicated by $p.n$, while $p + N$ is located at the splitter indicated by $p.node$. (Thus, p ascends the renaming tree in its exit section, while $p + N$ remains stationary until the exit section terminates.)

We define the *contention* of splitter i, denoted $C(i)$, as the number of processes p (shadow processes included) that are located at a splitter (or a child of a leaf splitter) in the subtree rooted at i. Consider a non-root splitter $i \geq 2$. If a process p in its entry section descends into splitter i, then $C(i)$ increases by two (because p and $p+N$ descend

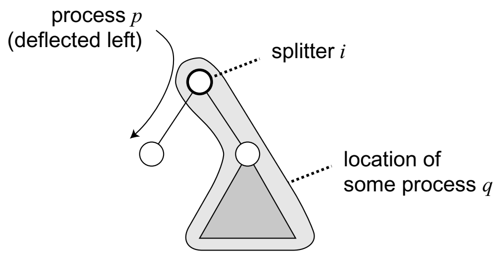


Figure 4.9: Deflection of a process p at splitter i. If p is deflected left (right), then some process q (or $q + N$) is located either at i or in the subtree rooted at i's right (left) child.

together). When p ascends from i to i's parent in its exit section, $C(i)$ decreases by one. Finally, when p finishes its exit section (by executing statement 41), both p and $p + N$ leave the renaming tree. Since $p + N$ has been located at p's acquired node (which is inside the subtree rooted at i), this reduces $C(i)$ by one again. On the other hand, $C(1)$ increases by two when p starts its execution, and decreases by two when p finishes its execution. Thus, $C(1)$ equals twice the actual point contention at any given state.

According to the following property, whenever a process p is deflected left or right, there exists another process (or a shadow process) q that causes p's deflection. (See Figure 4.9.) If p reaches a splitter at level l, then p is deflected l times, and hence there exist l other processes (or shadow processes) executing concurrently with p. From this, we can prove that $C(1) > l$ holds at some state.

(Of course, since these l processes may start and finish their execution at different times, it is not obvious that there should exist a *single* state satisfying $C(1) > l$, *i.e*, a state at which *all* of these l processes, or possibly some other set of l processes, are concurrently active. Property 8, given later, proves that such a state exists.)

Property 7: Consider a process p in its entry section. If p is deflected left (respectively, right) from a splitter i, then there exists another process (or shadow process) q that is concurrently located at splitter i, or in a subtree rooted at the right (respectively, left) child of i.

Proof: Process p may be deflected left or right by one of the following ways: **(i)** reading $Y[i] = (\textit{false}, 0)$ at statement 16, **(ii)** reading $X[i] \neq p$ at statement 19, **(iii)** reading $\textit{Acquired}[i] = \textit{true}$ at statement 20, and **(iv)** reading $\textit{Reset}[i] \neq p.y$ at statement 22. Among these four possibilities, only Case (i) deflects p right.

Case (i) may happen in two circumstances. First, a process q may have executed statement 17. In this case, q either has acquired splitter i, or has been deflected left. In either case, q is located at splitter i or in the subtree rooted at i's left child. Moreover, if q ascends to i's parent in its exit section, then q must either reopen splitter i by establishing $Y[i].\mathit{free} = \mathit{true}$ at statement 38[i], or read $\mathit{Round}[\mathit{Reset}[i].\mathit{rnd}] = \mathit{true}$ at statement 30[i]. However, the latter case may arise only if some other process q' has executed statement 21[i], in which case q' must be located at splitter i or in a subtree rooted at i's left child. Continuing in the same way, it follows that the following property continues to hold while $Y[i] = (\mathit{false}, 0)$ holds: *some process other than p is located at splitter i or in the subtree rooted at i's left child.*

Second, a process q may have executed statement 26. In this case, when p executes statement 16, either q is executing within statements 27[i]–38[i] (in which case q is located at splitter i), or it returned from *ReleaseNode* without executing statements 30[i]–40[i]. (Note that statement 38[i].q falsifies $Y[i] = (\mathit{false}, 0)$.) However, the latter case happens only if q reads $\mathit{Round}[\mathit{Reset}[i].\mathit{rnd}] = \mathit{true}$ at statement 30[i], which in turn happens only if (as shown above) there exists yet another process q' that has executed statement 21[i]. The rest of the reasoning is the same as the preceding paragraph.

Case (ii) may happen only if the following sequence of events happen: p executes statement 15, some process q executes statement 15[i] or 27[i], and then p executes statement 19. When q executes either statement, it is located at splitter i. (Of course, it is entirely possible that q has left splitter i by the time p executes statement 15. To guard against such a case, we define that p is actually deflected *at the instant* q executes statement 15[i] or 27[i], as explained shortly. A similar remark applies to Cases (iii) and (iv).)

Case (iii) may happen only if some process q has acquired splitter i. Thus, in this case, $q + N$ is located at splitter i.

Finally, consider Case (iv). Process p may reach statement 22 only if it has read $Y[i] = (\mathit{true}, r)$ at statement 16. Since *Reset* is always updated inside *ReleaseNode*, and since invocations of *ReleaseNode* do not overlap, it is easy to see that $Y[i] = (\mathit{true}, r)$ implies $Y[i] = \mathit{Reset}[i]$. Therefore, Case (iv) may happen only if the following sequence of events happen: p reads $Y[i] = (\mathit{true}, r)$ at statement 16, some process q executes statement 29[i] or 37[i], and then p executes statement 22. When q executes either statement, it is located at splitter i. $\square$

In particular, Property 7 implies the following: if p is deflected from a node h to its child i, then $C(i) < C(h)$ holds at that instant.

Note that the previous argument is somewhat simplified. In particular, we did not precisely define when p is deflected left or right. In order to formally prove contention sensitivity, we need to define p's location in a rather complicated manner. This is done in Appendix A by means of auxiliary variables $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$. (For example, assume that p executes statement $15[i]$ and stalls until after another process q executes statement $27[i]$. In this case, p is bound to be deflected right at statement $19[i]$, so we define that p's location changes when q writes $X[i] := q$.) For the sake of simplicity, we have ignored such details here.

Consider a process p in its entry section. At any given state t, if p is located in the subtree rooted at i, then we define $\mathsf{PC}[p, i]$, the *point contention of splitter i experienced by p*, as the maximum value of $C(i)$ since p descended into splitter i up to state t. In particular, when p moves down to splitter i, $\mathsf{PC}[p, i]$ is initialized to be $C(i)$. From that point onward, $\mathsf{PC}[p, i]$ tracks the point contention of the subtree rooted at i as seen by process p.

The following property states that, even if contention $C(i)$ may vary over time, the point contention $\mathsf{PC}[p, i]$ is always strictly lower than its parent's point contention $\mathsf{PC}[p, \lfloor i/2 \rfloor]$.

Property 8: If a process p in its entry section is located in a subtree rooted at splitter $i \geq 2$, then $\mathsf{PC}[p, i] < \mathsf{PC}[p, \lfloor i/2 \rfloor]$.

Proof: Define $h = \lfloor i/2 \rfloor$. First, consider the time when p descends into splitter i. (Note that $\mathsf{PC}[p, i]$ is not defined before this time.) By Property 7, we have $C(i) < C(h)$. By definition, $\mathsf{PC}[p, h] \geq C(h)$ holds. Since $\mathsf{PC}[p, i]$ is initialized to $C(i)$, the property follows.

Second, consider the case when $\mathsf{PC}[p, i]$ or $\mathsf{PC}[p, h]$ is changed while p is already inside the subtree rooted at i. Since $\mathsf{PC}[p, i]$ and $\mathsf{PC}[p, h]$ may only increase, it suffices to consider the case when $\mathsf{PC}[p, i]$ increases. However, this may happen only if some other process q descends into splitter i. By Property 7 again, at that time, $C(i) < C(h)$ holds. Therefore, if $\mathsf{PC}[p, i]$ is updated to the new value of $C(i)$, then we still have $\mathsf{PC}[p, h] \geq C(h) > C(i) = \mathsf{PC}[p, i]$, and hence the property follows. □

We now prove contention sensitivity. (For full proof, see invariant (I61) in Appendix A.) Assume that a process p has reached splitter i at level $l = \mathsf{lev}(i)$. By repeatedly applying Property 8 over all ancestors of i, we have

$$\begin{aligned} \mathsf{PC}[p,1] &= \mathsf{PC}[p, p.path[0].node] \\ &> \mathsf{PC}[p, p.path[1].node] \\ &\quad \vdots \\ &> \mathsf{PC}[p, p.path[p.level-1].node] \\ &> \mathsf{PC}[p, p.node] = \mathsf{PC}[p, i] \\ &> 0. \end{aligned}$$

Given the length of this sequence, we have $\mathsf{PC}[p, 1] > l$, which implies that p has experienced contention at least $l+1$ at some point since it started execution. It follows that Algorithm A-LS is contention-sensitive.

The space complexity of Algorithm A-LS is clearly $\Theta(N)$, if we ignore the space required to implement the `ENTRY` and `EXIT` routines. (Although each process has a $\Theta(\log N)$ *path* array, these arrays are actually unneeded, as simple calculations can be used to determine the parent and children of a splitter.) If the `ENTRY`/`EXIT` routines are implemented using Algorithm YA-N (Section 2.2.2), then the overall space complexity is actually $\Theta(N \log N)$. This is because in Algorithm YA-N, each process needs a distinct spin location for each level of the arbitration tree. However, as shown in Chapter 3, it is quite straightforward to modify the arbitration-tree algorithm so that each process uses the same spin location at each level of the tree. This modified algorithm (Algorithm F, illustrated in Figure 3.3) has $\Theta(N)$ space complexity. We conclude this section by stating our main theorem.

Theorem 4.1 *N-process mutual exclusion can be implemented under read/write atomicity with RMR time complexity $O(\min(k, \log N))$ and space complexity $\Theta(N)$.* □

4.2 Concluding Remarks

We have presented an adaptive algorithm for mutual exclusion under read/write atomicity in which all waiting is by local spinning. This is the first read/write algorithm that is adaptive under the RMR time complexity measure. Our algorithm has $\Theta(N)$ space complexity, which is clearly optimal.

In Chapter 5, we establish a lower bound of $\Omega(\log N/\log\log N)$ remote memory references for mutual exclusion algorithms based on reads, writes, or comparison primitives such as *test-and-set* or *compare-and-swap*. In Chapter 6, we also show that that it is impossible to construct an adaptive algorithm with $o(k)$ RMR time complexity. We conjecture that $\Omega(\log N)$ is a tight lower bound (under the RMR measure) for mutual exclusion algorithms under read/write atomicity, and that $\Omega(\min(k, \log N))$ is also a tight lower bound for adaptive mutual exclusion algorithms under read/write atomicity (in which case ALGORITHM A-LS of this chapter is optimal). We leave these questions for further study.

CHAPTER 5

Time-complexity Lower Bound for General Mutual Exclusion*

In this chapter, we establish a lower bound of $\Omega(\log N/\log\log N)$ RMRs (remote memory references) for N-process mutual exclusion algorithms based on reads, writes, or comparison primitives such as *test-and-set* and *compare-and-swap*.

Our lower bound is of importance for two reasons. First, this bound is within a factor of $\Theta(\log\log N)$ of being optimal, given Yang and Anderson's algorithm [84] (Algorithm YA-N, described in Section 2.2.2). Second, our lower bound suggests that it is likely that, from an asymptotic standpoint, comparison primitives are no better than reads and writes when implementing local-spin mutual exclusion algorithms. Thus, comparison primitives may not be the best choice to provide in hardware if one is interested in scalable synchronization.

As noted in Section 2.6, Anderson and Yang established trade-offs between time complexity and write- and access-contention for mutual exclusion algorithms in earlier work [22] (Theorems 2.10, 2.11). The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access by reading and/or writing) the same shared variable. These results imply that a trade-off between contention and time complexity exists even if coherent caching techniques are employed.

Yang and Anderson's lower bounds are meaningful only for algorithms with limited write- or access-contention. As noted in Section 2.6, Cypher [33] presented a lower

*The results presented in this chapter have been published in the following paper.
[17] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2002.

bound of $\Omega(\log\log N/\log\log\log N)$ under arbitrary access-contention (Theorem 2.16). His lower bound applies to algorithms using reads, writes, and comparison primitives such as *test-and-set* and *compare-and-swap*. (A *comparison primitive*, formally defined later in Section 5.1, conditionally updates a shared variable after first testing that its value meets some condition.) Given that primitives such as *fetch-and-add* and *fetch-and-store* can be used to implement mutual exclusion in $O(1)$ time, this result pointed to an unexpected weakness of *compare-and-swap*, which is still widely regarded as being one of the most useful of all synchronization primitives to provide in hardware.

In this chapter, we show that Cypher's lower bound can be improved to $\Omega(\log N/\log\log N)$. Thus, we have *almost* succeeded in establishing the optimality of Algorithm YA-N. Our result is stronger than Cypher's in that we allow atomic operations that may access many local shared variables (*i.e.*, variables that are statically associated to a process). Like Cypher's result, ours is applicable to algorithms that use comparison primitives (on remote variables), and is applicable to most DSM and CC systems. The only exception is a system with write-update caches[1] in which comparison primitives are supported, and with hardware capable of executing failed comparison primitives on cached remote variables without interconnection network traffic. We call such a system an LFCU ("Local Failed Comparison with write-Update") system. For LFCU systems, we show that $O(1)$ time complexity is possible using only *test-and-set* (the simplest of all comparison primitives).[2] While this result seemingly suggests that comparison primitives offer some advantages over reads and writes, it is worth noting that write-update protocols are almost never implemented in practice [68, page 721].

Our lower bound suggests that it is likely that, for non-LFCU systems, comparison primitives are no better from an asymptotic standpoint than reads and writes when implementing local-spin mutual exclusion algorithms. Moreover, the time complexity gap that exists in such systems between comparison primitives and primitives such as *fetch-and-add* and *fetch-and-store* is actually quite wide. Thus, comparison primitives

[1]Recall that, in a system with *write-update* caches, when a processor writes to a variable v that is also cached on other processors, a message is sent to these processors so that they can *update* v's value and maintain cache consistency.

[2]Although Cypher established a lower bound of $\Omega(\log\log N/\log\log\log N)$ for CC machines *with* comparison primitives, his cache-coherence model does not encompass write-update caches. According to his model, if a process p writes a variable v, then the first read of v by any other process q after p's write causes network traffic. Thus, if many processes read the value of v written by p, then each such read counts as a cache miss.

may not be the best choice to provide in hardware if one is interested in scalable synchronization.

The rest of this chapter is organized as follows. In Section 5.1, we present our model of atomic shared-memory systems. The key ideas of our lower bound proof are then sketched in Section 5.2. In Section 5.3, the proof is presented in detail. The $O(1)$ algorithm for LFCU systems mentioned above is then presented in Section 5.4. Concluding remarks appear in Section 5.5.

5.1 Definitions

In this section, we provide definitions pertaining to atomic shared-memory systems that will be used in the rest of this chapter and in Chapter 6. In the following subsections, we define our model of an atomic shared-memory system (Section 5.1.1), state the properties required of a mutual exclusion algorithm implemented within this model (Section 5.1.2), and present a categorization of events that allows us to accurately deduce the network traffic generated by an algorithm in a system with coherent caches (Section 5.1.3).

5.1.1 Atomic Shared-memory Systems

Our model of an atomic shared-memory system is similar to that used by Anderson and Yang [22].

An *atomic shared-memory system* $\mathcal{S} = (C, P, V)$ consists of a set of computations C, a set of processes P, and a set of variables V. A *computation* is a finite sequence of events. To complete the definition of an atomic shared-memory system, we must formally define the notion of an "event" and state the requirements to which events and computations are subject. This is done in the remainder of this subsection. As needed terms are defined, various notational conventions are also introduced that will be used in this chapter and Chapter 6.

Informally, an *event* is a particular execution of an atomic statement of some process that involves reading and/or writing of one or more variables. Each variable is *local* to at most one process and is *remote* to all other processes. (Note that we allow variables that are remote to *all* processes; thus, our model applies to both DSM and CC systems.) The locality relationship is static, *i.e.*, it does not change during a computation. A local variable may be shared; that is, a process may access local variables of other processes.

An *initial value* is associated with each variable. An event is *local* if it does not access any remote variables, and is *remote* otherwise.

Events, informally considered. Below, formal definitions pertaining to events are given; here, we present an informal discussion to motivate these definitions. An event is executed by a particular process, and may access at most one variable that is remote to that process (by reading, writing, or executing a comparison primitive), plus any number of local (shared) variables.[3] Thus, we allow arbitrarily powerful operations on local variables. Since our proof applies to systems with reads, writes, and comparison primitives, it is important to formally define what is a comparison primitive. We define a *comparison primitive* to be an atomic operation on a shared variable v expressible using the following pseudo-code.

```
$compare\_and\_fg(v, old, new)$
    $temp := v$;
    **if** $v = old$ **then** $v := f(old, new)$ **fi**;
    **return** $g(temp, old, new)$
```

For example, *compare-and-swap* can be defined by defining $f(old,\ new) = new$ and $g(temp,\ old,\ new) = old$. We call an execution of such a primitive a *comparison event*. As we shall see, our formal definition of a comparison event, which is given later in this section, generalizes the functionality encompassed by the pseudo-code above by allowing arbitrarily many local shared variables to be accessed.

As an example, assume that variables a, b, and c are local to process p and variables x and y are remote to p. Then, the following atomic statements by p are allowed in our model.

statement $s1$:	$a := a + 1;\ b := c + 1;$
statement $s2$:	$a := x;$
statement $s3$:	$y := a + b;$
statement $s4$:	*compare-and-swap*$(x, 0, b)$

For example, if every variable has an initial value of 0, and if these four statements are executed in order, then we will have the following four events.

[3]We do not distinguish between private and shared variables in our model. In an actual algorithm, some variables local to a process might be private and others shared.

$e1$: p reads 0 from a, writes 1 to a, reads 0 from c, and writes 1 to b;
/$*$ local event $*$/

$e2$: p reads 0 from x and writes 0 to a; /$*$ remote read from x $*$/

$e3$: p reads 0 from a, reads 1 from b, and writes 1 to y;
/$*$ remote write to y $*$/

$e4$: p reads 0 from x, reads 1 from b, and writes 1 to x
/$*$ comparison primitive execution on x $*$/

On the other hand, the following atomic statements by p are not allowed in our model, because $s5$ accesses two remote variables at once, and $s6$ and $s7$ cannot be expressed as a comparison primitive.

statement $s5$: $x := y$; /$*$ accesses two remote variables $*$/
statement $s6$: $a := x$; $x := 1$;
/$*$ *fetch-and-store* (*swap*) on a remote variable $*$/
statement $s7$: $x := x + b$ /$*$ *fetch-and-add* on a remote variable $*$/

Describing each event as in the preceding examples is inconvenient, ambiguous, and prone to error. For example, if statement $s7$ is executed when $x = 0 \ \wedge \ b = 1$ holds, then the resulting event can be described in the same way as $e4$ is. (Thus, $e4$ is allowed as an execution of $s4$, yet disallowed as an execution of $s7$.) In order to systematically represent the class of allowed events, we need a more refined formalism.

Definitions pertaining to events. An *event* e is denoted $[p, \mathsf{Op}, R, W]$, where $p \in P$ (the set of processes). We call Op the *operation* of event e, denoted $op(e)$. Op determines what kind of event e is, and can be one of the following: $\perp$, $\mathsf{read}(v)$, $\mathsf{write}(v)$, or $\mathsf{compare}(v, \alpha)$, where v is a variable in V and α is a value from the value domain of v. Informally, e can be a local event, a remote read, a remote write, or an execution of a comparison primitive. (The precise definition of these terms is given below.)

The sets R and W consist of pairs (v, α), where $v \in V$. This notation represents an event of process p that reads the value α from variable v for each element $(v, \alpha) \in R$, and writes the value α to variable v for each element $(v, \alpha) \in W$. Each variable in R is assumed to be distinct; the same is true for W. We define $Rvar(e)$, the set of variables read by e, to be $\{v\colon (v, \alpha) \in R\}$, and $Wvar(e)$, the set of variables written by e, to be $\{v\colon (v, \alpha) \in W\}$. We also define $var(e)$, the set of all variables accessed by e, to be $Rvar(e) \cup Wvar(e)$. We say that an event e *writes* (respectively, *reads*) a variable v if $v \in Wvar(e)$ (respectively, $v \in Rvar(e)$) holds, and that it *accesses* any variable that it writes or reads. We also say that a computation H *contains a write* (respectively, *read*) of v if H contains some event that writes (respectively, reads) v. Finally, we say that process p is the *owner* of $e = [p, \mathsf{Op}, R, W]$, denoted $\mathit{owner}(e) = p$. For brevity, we sometimes use e_p to denote an event owned by process p.

Our lower bound is dependent on the Atomicity property stated below. This assumption requires each remote event to be an atomic read operation, an atomic write operation, or a comparison-primitive execution.

Atomicity property: Each event $e = [p, \mathsf{Op}, R, W]$ must satisfy one of the conditions below.

- If $\mathsf{Op} = \bot$, then e does not access any remote variables. (That is, all variables in $\mathit{var}(e)$ are local to p.) In this case, we call e a *local event.*
- If $\mathsf{Op} = \mathsf{read}(v)$, then e reads exactly one remote variable, which must be v, and does not write any remote variable. (That is, $(v, \alpha) \in R$ holds for some α, v is not in $\mathit{Wvar}(e)$, and all other variables [if any] in $\mathit{var}(e)$ are local to p.) In this case, e is called a *remote read event.*
- If $\mathsf{Op} = \mathsf{write}(v)$, then e writes exactly one remote variable, which must be v, and does not read any remote variable. (That is, $(v, \alpha) \in W$ holds for some α, v is not in $\mathit{Rvar}(e)$, and all other variables [if any] in $\mathit{var}(e)$ are local to p.) In this case, e is called a *remote write event.*
- If $\mathsf{Op} = \mathsf{compare}(v, \alpha)$, then e reads exactly one remote variable, which must be v. We say that e is a *comparison event* in this case. Comparison events must be either successful or unsuccessful.
  - e is a *successful comparison event* if the following hold: $(v, \alpha) \in R$ (*i.e.*, e reads the value α from v), and $(v, \beta) \in W$ for some $\beta \neq \alpha$ (*i.e.*, e writes to v a value *different* from α).
  - e is an *unsuccessful comparison event* if e does not write v, *i.e.*, $v \notin \mathit{Wvar}(e)$ holds.

  In either case, e does not write any other remote variable. □

Our notion of an unsuccessful comparison event includes both comparison-primitive invocations that fail (*i.e.*, $v \neq \mathit{old}$ in the pseudo-code given for *compare_and_fg* above) and also those that do not fail but leave the remote variable that is accessed unchanged (*i.e.*, $v = old \ \wedge \ v = f(old, new)$). In the latter case, we simply assume that the remote variable v is not written. We categorize both cases as unsuccessful comparison events because this allows us to simplify certain cases in the proofs in Section 5.3 and Chapter 6. (On the other hand, we allow a remote write event on v to preserve the value of v, *i.e.*, to write the same value as v had before the event.)

Note that the Atomicity property allows arbitrarily powerful operations on local (shared) variables. For example, if variable v, ranging over $\{0, \ldots, 10\}$, is remote to process p, and arrays $a[1..10]$ and $b[1..10]$ are local to p, then an execution of the following statement is a valid event by p with operation $\mathsf{compare}(v, 0)$.

```
if $v = 0$ then
    $v := \left(\sum_{j=1}^{10} a[j]\right)$ mod 11;
    for $j := 1$ to 10 do $a[j] := b[j]$ od
else
    for $j := 1$ to $v$ do $b[j] := a[j] + v$ od
fi
```

In this case, $Wvar(e)$ is $\{v, a[1..10]\}$ if e reads $v = 0$ and writes a nonzero value (*i.e.*, e is a successful comparison event), $\{a[1..10]\}$ if e reads and writes $v = 0$, and $\{b[1..v]\}$ if e reads a value between 1 and 10 from v.

It is important to note that, saying that an event e_p writes (reads) a variable v is *not* equivalent to saying that e_p is a remote write (read) operation on v; e_p may also write (read) v if v is local to process p, or if p is a comparison event that accesses v.

We say that two events $e = [p, \mathsf{Op}, R, W]$ and $f = [q, \mathsf{Op}', R', W']$ are *congruent*, denoted $e \sim f$, if and only if the following conditions are met.

- $p = q$;
- $\mathsf{Op} = \mathsf{Op}'$, where equality means that both operations are the same *with the same arguments* (v and/or α).

Informally, two events are congruent if they execute the same operation on the same remote variable. For read and write events, the values read or written may be different. For comparison events, the values read or written (if successful) may be different, but the parameter α must be the same. (It is possible that a successful comparison operation is congruent to an unsuccessful one.) Note that e and f may access different *local* variables.

Definitions pertaining to computations. The definitions given until now have mostly focused on events. We now present requirements and definitions pertaining to computations.

The value of variable v at the end of computation H, denoted $\mathit{value}(v, H)$, is the last value written to v in H (or the initial value of v if v is not written in H). The last event to write to v in H is denoted $\mathit{writer\_event}(v, H)$,[4] and its owner is denoted $\mathit{writer}(v, H)$. If v is not written by any event in H, then we let $\mathit{writer}(v, H) = \perp$ and $\mathit{writer\_event}(v, H) = \perp$.

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, $\langle e, \ldots, f \rangle$ to denote a computation beginning with event e and ending with event f, and $\langle\rangle$ to denote the empty computation. We use $H \circ G$ to denote the computation obtained by concatenating computations H and G. An *extension* of computation H is a computation of which H is a prefix. For a computation H and a set of processes Y, $H \mid Y$ denotes the subcomputation of H that contains all events in H of processes in Y.[5] If G is a subcomputation of H, then $H - G$ is the computation obtained by removing all events in G from H. Computations H and G are *equivalent with respect to* Y if and only if $H \mid Y = G \mid Y$. A computation H is a Y*-computation* if and only if $H = H \mid Y$. For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes. For example, if $Y = \{p\}$, then we use $H \mid p$ to mean $H \mid \{p\}$ and p-computation to mean $\{p\}$-computation. Two computations H and G are *congruent*, denoted $H \sim G$, if either both H and G are empty, or if $H = \langle e \rangle \circ H'$ and $G = \langle f \rangle \circ G'$, where $e \sim f$ and $H' \sim G'$.

Until this point, we have placed no restrictions on the set of computations C of an atomic shared-memory system $\mathcal{S} = (C, P, V)$ (other than restrictions pertaining to the kinds of events that are allowed in an individual computation). The restrictions we require are as follows.

P1: If $H \in C$ and G is a prefix of H, then $G \in C$.
— *Informally, every prefix of a valid computation is also a valid computation.*

[4]Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other. (For simplicity, we do not extend our notion of an event to include an additional identifier for distinguishability.)

[5]The subcomputation $H \mid Y$ is not necessarily a valid computation in a given system $\mathcal{S}$, that is, an element of C. However, we can always consider $H \mid Y$ to be a computation in a technical sense, *i.e.*, it is a sequence of events.

P2: If $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \mid p = H \mid p$, and if $value(v, G) = value(v, H)$ holds for all $v \in Rvar(e_p)$, then $G \circ \langle e_p \rangle \in C$.
— *Informally, if two computations H and G are not distinguishable to process p, if p can execute event e_p after H, and if all variables in $Rvar(e_p)$ have the same values after H and G, then p can execute e_p after G.*

P3: If $H \circ \langle e_p \rangle \in C$, $G \in C$, and $G \mid p = H \mid p$, then $G \circ \langle e'_p \rangle \in C$ for some event e'_p such that $e_p \sim e'_p$.
— *Informally, if two computations H and G are not distinguishable to process p, and if p can execute event e_p after H, then p can execute a congruent operation after G.*

P4: For any $H \in C$, $H \circ \langle e_p \rangle \in C$ implies that $\alpha = value(v, H)$ holds, for all $(v, \alpha) \in Rvar(e_p)$.
— *Informally, only the last value written to a variable can be read.*

P5: For any $H \in C$, if both $H \circ \langle e_p \rangle \in C$ and $H \circ \langle e'_p \rangle \in C$ hold for two events e_p and e'_p, then $e_p = e'_p$.
— *Informally, each process is deterministic. This property is included in order to simplify bookkeeping in our proofs.*

Property P3 precisely defines the class of allowed events. In particular, if process p is enabled to execute a certain statement, then that statement must generate the same operation regardless of the execution of other processes. For example, if a is a local *shared* variable and x and y are remote variables, then the following statement is *not* allowed.

statement $s8$: **if** $a = 0$ **then** $x := 1$ **else** $y := 1$ **fi**

This is because the event generated by $s8$ may have either $\mathsf{write}(x)$ or $\mathsf{write}(y)$ as its operation, depending on the value possibly written to a by other processes.

5.1.2 Mutual-exclusion Systems

We now define a special kind of atomic shared-memory system, namely (atomic) mutual exclusion systems, which are our main interest. An *atomic mutual exclusion system* $\mathcal{S} = (C, P, V)$ is an atomic shared-memory system that satisfies the properties below.

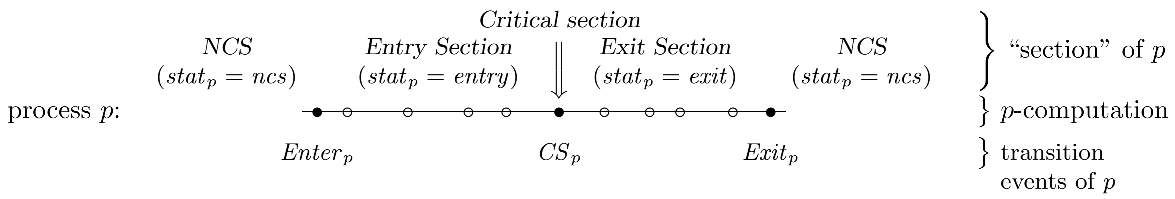


Figure 5.1: Transition events of an atomic mutual exclusion system. In this figure, NCS stands for "noncritical section," a circle (○) represents a non-transition event, and a bullet (●) represents a transition event.

Each process p has a local auxiliary variable $stat_p$ that represents which section in the mutual exclusion algorithm p is currently in: $stat_p$ ranges over *ncs* (for noncritical section), *entry*, or *exit*, and is initially *ncs*. (For simplicity, we assume that each critical-section execution is vacuous.) Process p also has three "dummy" auxiliary variables ncs_p, $entry_p$, and $exit_p$. These variables are accessed only by the following events.

$$\begin{aligned} Enter_p &= [\,\mathsf{write}(entry_p),\ \{\},\ \{(stat_p, entry),\ (entry_p, 0)\},\ p\,] \\ CS_p &= [\,\mathsf{write}(exit_p),\ \{\},\ \{(stat_p, exit),\ (exit_p, 0)\},\ p\,] \\ Exit_p &= [\,\mathsf{write}(ncs_p),\ \{\},\ \{(stat_p, ncs),\ (ncs_p, 0)\},\ p\,] \end{aligned}$$

Event $Enter_p$ cause p to transit from its noncritical section to its entry section. Event CS_p cause p to transit from its entry section to its exit section.[6] Event $Exit_p$ cause p to transit from its exit section to its noncritical section. This behavior is depicted in Figure 5.1.

We define variables $entry_p$, $exit_p$, and ncs_p to be remote to all processes. This assumption allows us to simplify bookkeeping, because it implies that each of $Enter_p$, CS_p, and $Exit_p$ is congruent only to itself. (This is the sole purpose of defining these three variables.)

The allowable transitions of $stat_p$ are as follows: for all $H \in C$,

$$\begin{array}{rcl} H \circ \langle Enter_p \rangle \in C & \text{if and only if} & value(stat_p, H) = ncs; \\ H \circ \langle CS_p \rangle \in C & \text{only if} & value(stat_p, H) = entry; \\ H \circ \langle Exit_p \rangle \in C & \text{only if} & value(stat_p, H) = exit. \end{array}$$

In our proof, we only consider computations in which each process enters and then exits its critical section at most once. Thus, we henceforth assume that each compu-

[6]Each critical-section execution of p is captured by the single event CS_p, so $stat_p$ changes directly from *entry* to *exit*.

tation contains at most one Enter_p event for each process p. In addition, an atomic mutual exclusion system is required to satisfy the following.

Exclusion: For all $H \in C$, if both $H \circ \langle CS_p \rangle \in C$ and $H \circ \langle CS_q \rangle \in C$ hold, then $p = q$.

Progress: Given $H \in C$, define $X = \{q \in P\colon \mathit{value}(\mathit{stat}_q, H) \neq \mathit{ncs}\}$. If X is nonempty, then there exists an X-computation G such that $H \circ G \circ \langle e_p \rangle \in C$, where $p \in X$ and e_p is either CS_p (if $\mathit{value}(\mathit{stat}_p, H) = \mathit{entry}$) or Exit_p (if $\mathit{value}(\mathit{stat}_p, H) = \mathit{exit}$).

The Exclusion property is equivalent to (mutual) exclusion, which was informally defined in Chapter 1. Although we assume that each critical-section execution is vacuous, we can certainly "augment" the algorithm by replacing each event CS_p by a set of events that represents p's critical-section execution. If two events CS_p and CS_q are simultaneously "enabled" after a computation H, then we can interleave the critical-section executions of p and q, thus violating mutual exclusion. The Exclusion property states that such a situation does not arise.

The Progress property is implied by livelock-freedom, although it is strictly weaker than livelock-freedom. In particular, it allows the possibility of infinitely extending H such that no active process p executes CS_p or Exit_p. This weaker formalism is sufficient for our purposes.

5.1.3 Cache-coherent Systems

On cache-coherent (CC) shared-memory systems, some remote-variable accesses may be handled locally, without causing interconnection network traffic. Our lower-bound proofs apply to such systems without modification. This is because we do not count every remote event, but only certain "critical" events that generate cache misses. (Actually, as explained below, some events that we consider critical might not generate cache misses in certain system implementations, but this has no asymptotic impact on our proof.)

Precisely defining the class of such events in a way that is applicable to the myriad of cache implementations that exist is exceedingly difficult. We partially circumvent this problem by assuming idealized caches of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable. Note that, in practice, cache size and associativity limitations should only *increase* the number of cache misses. In addition, in order to keep the proof manageable, we allow cache misses to be both undercounted *and* overcounted. In particular, as explained below, in any realistic cache system, at least a constant fraction (but not necessarily all) of all critical events generate cache misses. Thus, a single cache miss may be associated with $\Theta(1)$ critical events, resulting in overcounting up to a constant factor. Note that this overcounting has no effect on our asymptotic lower bound. Also, an event that generates a cache miss may be considered noncritical, resulting in undercounting, which may be of more than a constant factor. Note that this undercounting can only strengthen our asymptotic lower bound. Therefore, an asymptotic lower bound on the number of critical events is also an asymptotic lower bound on the number of actual cache misses.

Our definition of a critical event is given below. This definition is followed by a rather detailed explanation in which various kinds of caching protocols are considered.

Definition: Let $\mathcal{S} = (C, P, V)$ be an atomic mutual exclusion system. Let e_p be an event in $H \in C$. Then, we can write H as $F \circ \langle e_p \rangle \circ G$, where F and G are subcomputations of H. We say that e_p is a *critical event* in H if and only if one of the following conditions holds:

Transition event: e_p is one of Enter_p, CS_p, or Exit_p.

Critical read: There exists a variable v, remote to p, such that $\mathit{op}(e_p) = \mathsf{read}(v)$ and $F \mid p$ does not contain a read from v.
— *Informally, e_p is the first event of p that reads v in H.*

Critical write: There exists a variable v, remote to p, such that e_p is a remote write event on v (*i.e.*, $\mathit{op}(e_p) = \mathsf{write}(v)$), and $\mathit{writer}(v, F) \neq p$.
— *Informally, e_p is a remote write event on v, and either e_p is the first event that writes to v in H (i.e., $\mathit{writer}(v, F) = \perp$), or e_p overwrites a value that was written by another process.*

Critical successful comparison: There exists a variable v, remote to p, such that e_p is a successful comparison event on v (*i.e.*, $\mathit{op}(e_p) = \mathsf{compare}(v, \alpha)$ for some value of α and $v \in \mathit{Wvar}(e_p)$), and $\mathit{writer}(v, F) \neq p$.
— *Informally, e_p is a successful comparison event on v, and either e_p is the first event that writes to v in H (i.e., $\mathit{writer}(v, F) = \perp$), or e_p overwrites a value that was written by another process.*

Critical unsuccessful comparison: There exists a variable v, remote to p, such that e_p is an unsuccessful comparison event on v (*i.e.*, $op(e_p) = \mathsf{compare}(v, \alpha)$ for some value of α and $v \notin Wvar(e_p)$), $writer(v, F) \neq p$, and either
(i) $F \mid p$ does not contain an unsuccessful comparison event on v, or
(ii) F can be written as $F_1 \circ \langle f_q \rangle \circ F_2$, where $f_q = writer\_event(v, F)$, such that $F_2 \mid p$ does not contain an unsuccessful comparison event on v.
— *Informally, e_p must read the initial value of v (if $writer(v, F) = \perp$) or a value that is written by another process q. Moreover, either* **(i)** *e_p is the first unsuccessful comparison on v by p in H, or* **(ii)** *e_p is the first such event by p after some other process has written to v (via f_q).*[7] □

Note that state transition events do *not* actually cause cache misses; these events are defined as critical events because this allows us to combine certain cases in the proofs that follow. A process executes only three transition events per critical-section execution, so defining transition events as critical does not affect our asymptotic lower bound.

Note that it is possible that the first read of v by p, the first write or successful comparison event on v by p, *and* the first unsuccessful comparison event on v by p (*i.e.*, Case (i) in the definition above) are all considered critical. Depending on the system implementation, the second and third of these events to occur might not generate a cache miss. However, even in such a case, the first such event will always generate a cache miss, and hence at least a third of all such "first" critical events will actually incur real interconnect traffic. Hence, considering all of these events to be critical has no asymptotic impact on our lower bound.

All caching protocols are based on either a *write-through* or a *write-back* scheme. In a write-through scheme, all writes go directly to shared memory. In a write-back scheme, a remote write to a variable v may create a cached copy of v, so that subsequent writes to v do not cause cache misses. With either scheme, if cached copies of v exist on other processors when such a write occurs, then to ensure consistency, these cached copies must be either *invalidated* or *updated*. In the rest of this subsection, we consider in some detail the question of whether our notion of a critical write and a critical

[7]This definition is more complicated than those for critical writes and successful comparisons because an unsuccessful comparison event on v by p does not actually write v. Thus, if a sequence of such events is performed by p while v is not written by other processes, then only the first such event should be considered critical.

comparison is reasonable under the various caching protocols that arise from these definitions.

First, consider a system in which there are no comparison events, in which case it is enough to consider only critical write events. If a write-through scheme is used, then all remote write events cause interconnect traffic, so consider a write-back scheme. In this case, a write e_p to a remote variable that is not the first write to v by p is considered critical only if $\mathit{writer}(v, F) = q$ holds for some $q \neq p$, which implies that v is stored in a local cache line of process q. (Since all caches are assumed to be infinite, $\mathit{writer}(v, F) = q$ implies that q's cached copy of v has not been invalidated.) In such a case, e_p must either invalidate or update the cached copy of v (depending on the means for ensuring consistency), thereby generating interconnect traffic.

Next, consider comparison events. A successful comparison event on a remote variable v writes a new value to v. Thus, the reasoning given above for ordinary writes applies to successful comparison events as well. This leaves only unsuccessful comparison events. Recall that an unsuccessful comparison event on a remote variable v does not actually write v. Thus, the reasoning above does *not* apply to such events.

In the remainder of this discussion, let e_p denote an unsuccessful comparison event on a remote variable v, where Case (ii) in the definition applies. Then, some other process q writes to v (via a write or successful comparison event, or even a local, read, or unsuccessful comparison event, if v is local to q) prior to e_p but after p's most recent unsuccessful comparison event on v, and also after p's most recent successful comparison and/or remote write event on v. Consider the interconnect traffic generated, assuming an invalidation scheme for ensuring cache consistency. In this case, p's previous cached copy of v is invalidated prior to e_p, so e_p must generate interconnect traffic in order to read the current value of v, unless an earlier read of v by p (after q's write) exists. Thus, e_p fails to generate interconnect traffic only if there is an earlier read of v by p (after q's write), say f_p, that does. Note that f_p is either a "first" read of v by p or a noncritical read. The former case may happen at most once per remote variable; in the latter case, we can "charge" the interconnect traffic generated by f_p to e_p.

The last possibility to consider is that of an unsuccessful comparison event e_p implemented within a caching protocol that uses updates to ensure consistency. In this case, q's write in the scenario above updates p's cached copy, and hence e_p may not generate interconnect traffic. (Note that, for interconnect traffic to be avoided in this case, the hardware must be able to distinguish a failed comparison event on a cached variable from a successful comparison event or a failed comparison on a non-cached variable.) Therefore, our lower bound does *not* apply to a system that uses updates to ensure consistency and that has the ability to execute failed comparison events on cached variables without generating interconnect traffic. (If an update scheme is used, but the hardware is incapable of avoiding interconnect traffic when executing such failed comparison events, then our lower bound obviously still applies.) Such systems were termed LFCU ("Local Failed Comparison with write-Update") systems earlier in this chapter. An algorithm with $O(1)$ time complexity in such systems is presented in Section 5.4. This algorithm shows that LFCU systems fundamentally *must* be excluded from our proof.

As a final comment on our notion of a critical event, notice that whether an event is considered critical depends on the particular computation that contains the event, specifically the prefix of the computation preceding the event. Therefore, when saying that an event is (or is not) critical, the computation containing the event must be specified.

5.2 Lower-bound Proof Strategy

In Section 5.3, we show that for any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation H such that some process p executes $\Omega(\log N/ \log\log N)$ critical events to enter and exit its critical section, where $N = |P|$. In this section, we sketch the key ideas of the proof.

5.2.1 Process Groups and Regular Computations

Our proof focuses on a special class of computations called "regular" computations. A regular computation consists of events of two groups of processes, "active processes" and "finished processes." Informally, an active process is a process in its entry section, competing with other active processes; a finished process is a process that has executed its critical section once, and is in its noncritical section. (Recall that we consider only computations in which each process executes is critical section at most once.) These properties follow from Condition RF4, given later in this section.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, and H be a computation in C. We define $\mathrm{Act}(H)$, the set of *active processes* in H, and $\mathrm{Fin}(H)$, the set of *finished* *processes* in H, as follows.

$$\begin{aligned}\mathrm{Act}(H) &= \{p \in P\colon H \,|\, p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is } \mathit{not} \text{ in } H\} \\ \mathrm{Fin}(H) &= \{p \in P\colon H \,|\, p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is in } H\}\end{aligned}$$

□

In Section 5.2.2, a detailed overview of our proof is given. Here, we give cursory overview, so that the definitions that follow will make sense. Initially, we start with a regular computation in which all the processes in P are active. The proof proceeds by inductively constructing longer and longer regular computations, until the desired lower bound is attained. The regularity condition defined below ensures that no participating process has "knowledge" of any other process that is active.[8] This has two consequences: we can "erase" any active process (*i.e.*, remove its events from the computation) and still get a valid computation; "most" active processes have a "next" non-transition critical event. In each induction step, we append to each of the n active processes (except at most one) one next critical event. These next critical events may introduce unwanted information flow, *i.e.*, these events may cause an active process to acquire knowledge of another active process, resulting in a non-regular computation. Informally, such information flow is problematic because an active process p that learns of another active process may start busy waiting. If p busy waits via a local spin loop, then it might *not* execute any more critical events, in which case the induction fails.

In some cases, we can eliminate all information flow by simply erasing some active processes. Sometimes erasing alone does not leave enough active processes for the next induction step. In this case, we partition the active processes into two categories: "invisible" processes and "promoted" processes. The invisible processes (that are not erased — see below) will constitute the set of active processes for the next regular computation in the induction. No process is allowed to have knowledge of another process that is invisible. The promoted processes are processes that have been selected to "roll forward." A process that is rolled forward finishes executing its entry, critical, and exit sections, and returns to its noncritical section. (Both of these techniques, erasing and rolling forward, have been used previously to prove other lower bounds related to the mutual exclusion problem (see Section 2.6), as well as several other lower bounds for concurrent systems [3, 79].) Processes *are* allowed to have knowledge of promoted or finished processes. Although invisible processes may have knowledge of

[8]A process p has knowledge of another process q if p has read from some variable a value that is written either by q or another process that has knowledge of q.

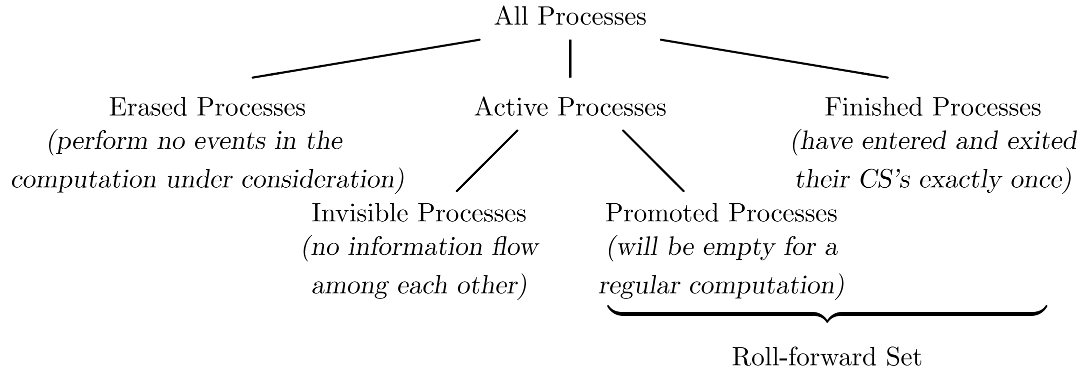

All Processes
Erased Processes
(perform no events in the computation under consideration)
Active Processes
Finished Processes
(have entered and exited their CS's exactly once)
Invisible Processes
(no information flow among each other)
Promoted Processes
(will be empty for a regular computation)
Roll-forward Set

Figure 5.2: Process groups.

promoted processes, once all promoted processes have finished execution, the regularity condition holds again (*i.e.*, all active processes are invisible). The various process groups we consider are depicted in Figure 5.2 (the roll-forward set is discussed below).

The promoted and finished processes together constitute a "roll-forward set," which must meet Conditions RF1–RF5 below. Informally, Condition RF1 ensures that an invisible process is not known to any other processes; RF2 and RF3 bound the number of possible conflicts caused by appending a critical event; RF4 ensures that the invisible, promoted, and finished processes behave as explained above; RF5 ensures that we can erase any invisible process, maintaining that critical events (that are not erased) remain critical.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, H be a computation in C, and RFS be a subset of P such that $\mathrm{Fin}(H) \subseteq RFS$ and $H \mid p \neq \langle\rangle$ for each $p \in RFS$. We say that RFS is a valid *roll-forward set* (*RF-set*) of H if and only if the following conditions hold.

RF1: Assume that H can be written as $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$.[9] If $p \neq q$ and there exists a variable $v \in Wvar(e_p) \cap Rvar(f_q)$ such that F does not contain a write to v (*i.e.*, $writer\_event(v, F) = \perp$), then $p \in RFS$ holds.
— *Informally, if a process p writes to a variable v, and if another process q reads that value from v without any intervening write to v, then $p \in RFS$ holds.*

RF2: For any event e_p in H and any variable v in $var(e_p)$, if v is local to another process q ($\neq p$), then either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq RFS$ holds.

[9]Here and in similar sentences hereafter, we are considering *every* way in which H can be so decomposed. That is, any pair of events e_p and f_q inside H such that e_p comes before f_q defines a decomposition of H into $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$, and RF1 must hold for any such decomposition.

— *Informally, if a process p accesses a variable that is local to another process q, then either q is not an active process in H, or both p and q belong to the roll-forward set RFS. Note that this condition does not distinguish whether q actually accesses v or not, and conservatively requires q to be in RFS (or erased) even if q does not access v. This is done in order to simplify bookkeeping.*

RF3: Consider a variable $v \in V$ and two different events e_p and f_q in H. Assume that both p and q are in $\mathrm{Act}(H)$, $p \neq q$, there exists a variable v such that $v \in \mathit{var}(e_p) \cap \mathit{var}(f_q)$, and there exists a write to v in H. Then, $\mathit{writer}(v, H) \in RFS$ holds.
— *Informally, if a variable v is accessed by more than one processes in* $\mathrm{Act}(H)$, *then the last process in H to write to v (if any) belongs to RFS.*

RF4: For any process p such that $H \mid p \neq \langle\rangle$,

$$\mathit{value}(\mathit{stat}_p,\ H) = \begin{cases} \mathit{entry} & \text{if } p \in \mathrm{Act}(H) - RFS, \\ \mathit{entry} \text{ or } \mathit{exit} & \text{if } p \in \mathrm{Act}(H) \cap RFS, \\ \mathit{ncs} & \text{otherwise } (\textit{i.e.}, p \in \mathrm{Fin}(H)). \end{cases}$$

Moreover, if $p \in \mathrm{Fin}(H)$, then the last event by p in H is Exit_p.
— *Informally, if a process p participates in H ($H \mid p \neq \langle\rangle$), then at the end of H, one of the following holds:* **(i)** *p is in its entry section and has not yet executed its critical section ($p \in \mathrm{Act}(H) - RFS$);* **(ii)** *$p$ is in the process of "rolling forward" and is in its entry or exit section ($p \in \mathrm{Act}(H) \cap RFS$); or* **(iii)** *$p$ has already finished its execution and is in its noncritical section (i.e., $p \in \mathrm{Fin}(H)$).*

RF5: For any event e_p in H, if e_p is a critical write or a critical comparison in H, then e_p is also a critical write or a critical comparison in $H \mid (\{p\} \cup RFS)$.
— *Informally, if an event e_p in H is a critical write or a critical comparison, then it remains critical if we erase all processes not in RFS and different from p.* □

Condition RF5 is used to show that the property of being a critical write/comparison is conserved when considering certain related computations. Recall that, if e_p is not the first event by p to write to v, then for it to be critical, there must be a write to v by another process q in the subcomputation between p's most recent write (via a remote write or a successful comparison event) and event e_p. Similarly, if e_p is not the first unsuccessful comparison by p on v, then for it to be critical, there must be a write to v by another process q in the subcomputation between p's most recent unsuccessful comparison on v and event e_p. RF5 ensures that if q is not in RFS, then other process q' exists that *is* in RFS and that writes to v in the subcomputation in question.

Note that a valid RF-set can be "expanded": if RFS is a valid RF-set of computation H, then any set of processes that participate in H, provided that it is a superset of RFS, is also a valid RF-set of H.

The invisible and promoted processes (which partition the set of active processes) are defined as follows.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, H be a computation in C, and RFS be a valid RF-set of H. We define $\mathrm{Inv}_{RFS}(H)$, the set of *invisible processes* in H, and $\mathrm{Pmt}_{RFS}(H)$, the set of *promoted processes* in H, as follows.

$$\begin{aligned}\mathrm{Inv}_{RFS}(H) &= \mathrm{Act}(H) - RFS\\ \mathrm{Pmt}_{RFS}(H) &= \mathrm{Act}(H) \cap RFS\end{aligned} \qquad \square$$

For brevity, we often omit the specific RF-set if it is obvious from the context, and simply use the notation $\mathrm{Inv}(H)$ and $\mathrm{Pmt}(H)$. Finally, the regularity condition can be defined as "all the processes we wish to roll forward have finished execution."

Definition: A computation H in C is *regular* if and only if $\mathrm{Fin}(H)$ is a valid RF-set of H. $\square$

5.2.2 Detailed Proof Overview

The overall structure of our proof is depicted in Figure 5.3. Initially, we start with H_1, in which $\mathrm{Act}(H_1) = P$, $\mathrm{Fin}(H_1) = \{\}$, and each process p has one critical event, $Enter_p$. We inductively construct longer and longer regular computations until our lower bound is met. At the j^{th} induction step, we consider a computation H_j such that $\mathrm{Act}(H_j)$ consists of n ($\leq N$) processes, each of which executes j critical events in H_j. We show that if $j > c \log n$, where c is a fixed constant, then the lower bound has already been attained. Thus, in the inductive step, we assume $j \leq c \log n$. Based on the existence of H_j, we construct a regular computation H_{j+1} such that $\mathrm{Act}(H_{j+1})$ consists of $\Omega(n/\log^2 n)$ processes,[10] each of which executes $j+1$ critical events in H_{j+1}. The

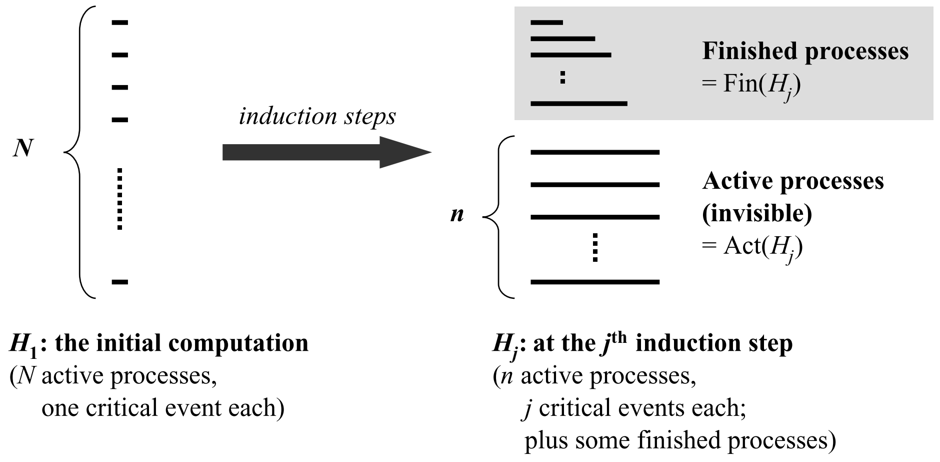


Figure 5.3: Induction steps. In this and subsequent figures, a computation is depicted as a collection of bold horizontal lines, with each line representing the events of a single process.

construction method, formally described in Lemma 5.7 (in Section 5.3), is explained below.

Since computation H_j is regular, no active process has knowledge of other active processes. Therefore, we can "erase" any active process and still get a valid computation (Lemma 5.1). Moreover, if we choose an active process p and erase all other active processes, then by the Progress property, p eventually executes CS_p, as shown in Figure 5.4. We claim that every process p in $\mathrm{Act}(H_j)$, except at most one, executes at least one additional critical event before it executes CS_p. This claim is formally stated and proved in Lemma 5.5; here we give an informal explanation.

Assume, to the contrary, that we have two distinct processes, p and q, each of which may execute its CS event, if executed alone as shown in Figure 5.4, by first executing only noncritical events. It can be shown that noncritical events of invisible processes cannot cause any information flow among these processes (Lemma 5.4). That is, an invisible process cannot become "visible" as a result of executing noncritical events. In particular, a local event of process p cannot cause information flow, because, by RF2, no other invisible process can access p's local variables. In addition, a remote read event of v by p is noncritical only if p has already read v. If another process q has written v after p's last read, then by RF3, the last process to write v is in RFS, *i.e.*, it is not invisible. Similar arguments apply to remote write or comparison events.

[10]Recall that we use $\log^2 n$ to denote $(\log n)^2$ (see page 16).

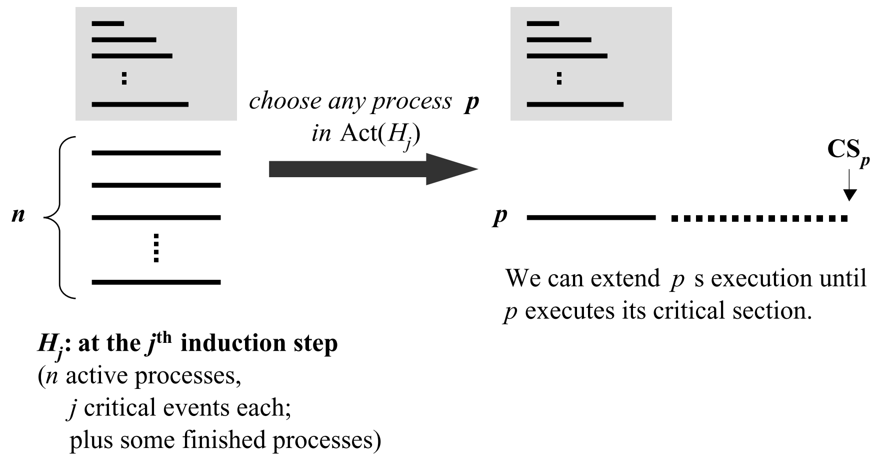

choose any process p in Act(H_j)
n
p
CS_p
We can extend p s execution until p executes its critical section.
H_j: at the j^{th} induction step
(n active processes,
j critical events each;
plus some finished processes)

Figure 5.4: Extending a regular computation. For each active (invisible) process p in a regular computation, if p runs in isolation (*i.e.*, with only finished processes), p eventually executes CS_p.

It follows that process p is unaware of q until it executes its critical section, and *vice versa*. Therefore, we can let both p and q execute *concurrently* after $H_j \mid (\{p, q\} \cup \mathrm{Fin}(H_j))$ — in particular, we can append $F_p \circ F_q$ (or $F_q \circ F_p$), constructing a computation that may be followed by *both* CS_p and CS_q, clearly violating the Exclusion property.[11]

Thus, among the n processes in $\mathrm{Act}(H_j)$, at least $n - 1$ processes can execute an additional critical event before entering its critical section. We call these events "next" critical events, and denote the corresponding set of processes by Y.

The processes in Y collectively execute at most nj critical events in H_j. Let $\mathcal{E}$ be the set of all these events plus all the next critical events. Since we assumed $j \le c \log n$, we have $|\mathcal{E}| \le nj + n \le (c \log n + 1)n$. Among the variables accessed by these events, we identify V_{HC}, the set of variables that experience "high contention," as those that are remotely accessed by at least $d \log^2 n$ critical events in $\mathcal{E}$, where d is another constant to be specified. Because of the Atomicity property, each event can access at most one remote variable, so we have $|V_{\mathrm{HC}}| \le (c \log n + 1)n/(d \log^2 n) \le (c+1)n/(d \log n)$. Next, we partition the processes in Y depending on whether their next critical events access a variable in V_{HC}:

$$\begin{aligned} P_{\mathrm{HC}} &= \{y \in Y\colon y\text{'s next critical event accesses some variable in } V_{\mathrm{HC}}\}, \\ P_{\mathrm{LC}} &= Y - P_{\mathrm{HC}}. \end{aligned}$$

[11]It is crucial that both F_p and F_q are free of critical events. For example, if F_q contains a critical event, then it may read a variable that is written by p in F_p, or write a variable that is read by p in F_p. In the former case, $F_p \circ F_q$ causes information flow; in the latter, $F_q \circ F_p$ does.

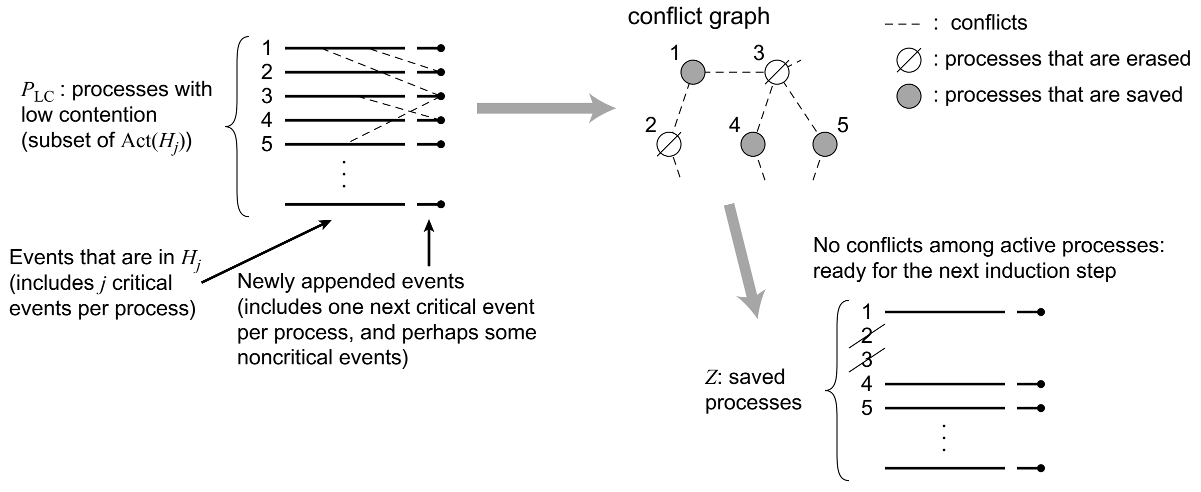

P_{LC} : processes with low contention (subset of $\mathrm{Act}(H_j)$)
1
2
3
4
5
Events that are in H_j (includes j critical events per process)
Newly appended events (includes one next critical event per process, and perhaps some noncritical events)
conflict graph
1
3
2
4
5
--- : conflicts
: processes that are erased
: processes that are saved
No conflicts among active processes: ready for the next induction step
Z: saved processes
1
2
3
4
5

Figure 5.5: Erasing strategy. For simplicity, processes in $\mathrm{Fin}(H_j)$ are not shown in the remaining figures.

Because H_j is regular and $Y \subseteq \mathrm{Act}(H_j)$, we can erase any process in Y. Hence, we can erase the smaller of P_{HC} and P_{LC} and still have $\Omega(n)$ remaining active processes. We consider these cases separately.

Erasing strategy. Assume that we erased P_{HC} and saved P_{LC}. This situation is depicted in Figure 5.5. In order to construct a regular computation, we want to eliminate any information flow. There are two cases to consider.

If p's next critical event reads a value that is written by some other active process q, then information flow clearly arises, and hence must be eliminated, by erasing either p or q. The other case is more subtle: if p's next critical event *overwrites* a value that is written by q, then although no information flow results, problems may arise later. In particular, assume that, in a later induction step, yet another process r reads the value written by p. In this case, simply erasing p does not eliminate all information flow, because then r would read a value written by q instead. In order to simplify bookkeeping, we simply assume that a conflict arises whenever any next critical event accesses any variable that is accessed by a critical event (past or next) of any other processes. That is, we consider all four possibilities, write followed by read, write followed by write, read followed by read, and read followed by write, to be conflicts.

Define V_{LC} as the set of variables remotely accessed by the next critical events of P_{LC}. Then clearly, every variable in V_{LC} is a "low contention" variable, and hence is accessed by at most $d \log^2 n$ different critical events (and, hence, different processes). Therefore,

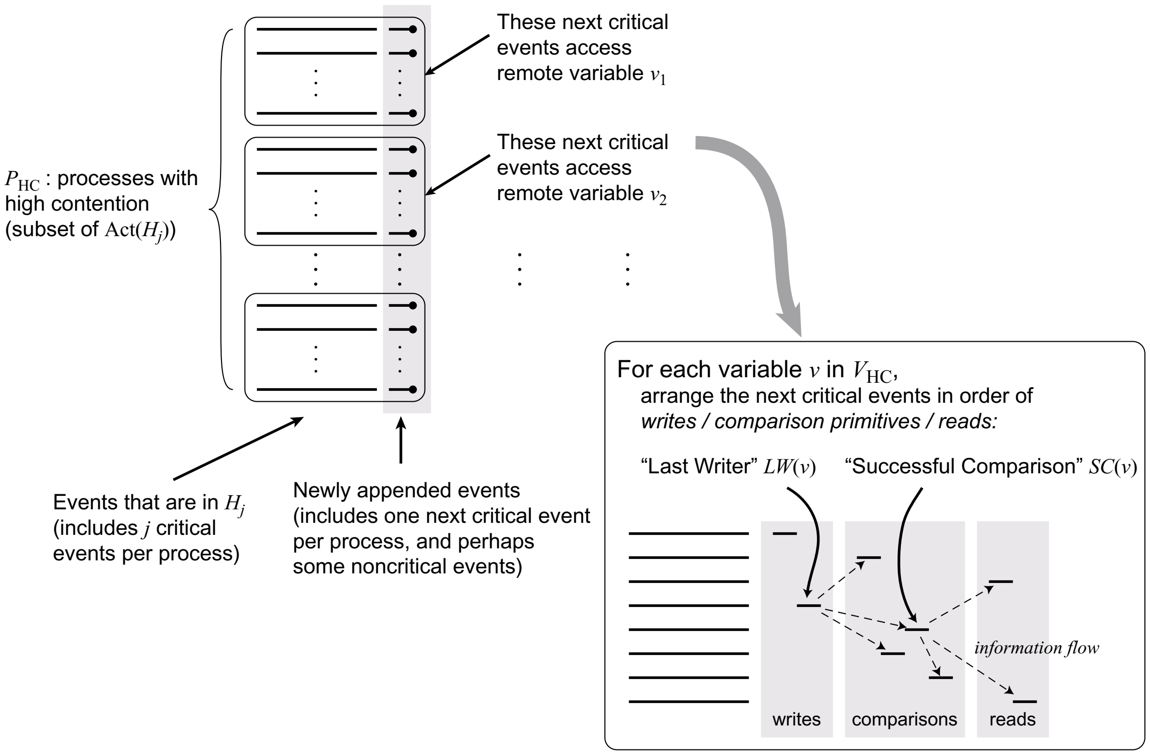

These next critical events access remote variable v_1
These next critical events access remote variable v_2
P_{HC} : processes with high contention (subset of Act(H_j))
Events that are in H_j (includes j critical events per process)
Newly appended events (includes one next critical event per process, and perhaps some noncritical events)
For each variable v in V_{HC}, arrange the next critical events in order of writes / comparison primitives / reads:
"Last Writer" $LW(v)$
"Successful Comparison" $SC(v)$
information flow
writes
comparisons
reads

Figure 5.6: Roll-forward strategy. Part I.

the next event by a process in P_{LC} can conflict with at most $d \log^2 n$ processes. By generating a "conflict graph" and applying Turán's theorem (Theorem 5.1), we can find a set of processes Z such that $|Z| = \Omega(n/\log^2 n)$, and among the processes in Z, there are no conflicts. By retaining Z and erasing all other active processes, we can eliminate all conflicts. Thus, we can construct H_{j+1}.

Roll-forward strategy. Assume that we erased P_{LC} and saved P_{HC}. This situation is depicted in Figure 5.6. In this case, the erasing strategy does not work, because eliminating all conflicts will leave us with at most $|V_{HC}|$ processes, which may be $o(n/\log^2 n)$.

Every next event by a process in P_{HC} accesses a variable in V_{HC}. For each variable $v \in V_{HC}$, we arrange the next critical events that access v by placing write, comparison, and read events in that order. Then, all next write events of v, except for the last one, are overwritten by subsequent writes, and hence cannot create any information flow. (That is, even if some other process later reads v, it cannot gather any information of these "next" writers, except for the last one.) Furthermore, we can arrange comparison events such that at most one of them succeeds, as follows.

Assume that the value of v is α after all the next write events are executed. We first append all comparison events with an operation that can be written as $\mathsf{compare}(v, \beta)$ such that $\beta \neq \alpha$. These comparison events must fail. We then append all the remaining comparison events, namely, events with operation $\mathsf{compare}(v, \alpha)$. The first successful event among them (if any) changes the value of v. Thus, all subsequent comparison events must fail.

Thus, among the next events accessing some such $v \in V_{\mathrm{HC}}$, the only information flow that arises is from the "last writer" event $LW(v)$ and from the "successful comparison" event $SC(v)$ to all other next comparison and read events of v. We define $\overline{G}$ to be the resulting computation with the next critical events arranged as above, and define the RF-set RFS as $\{LW(v), SC(v)\colon v \in V_{\mathrm{HC}}\} \cup \mathrm{Fin}(H_j)$. Then, by definition, the set of promoted processes $\mathrm{Pmt}(\overline{G})$ consists of $LW(v)$ and $SC(v)$ for each $v \in V_{\mathrm{HC}}$. Because $|V_{\mathrm{HC}}| \leq (c+1)n/(d \log n)$, we have $|\mathrm{Pmt}(\overline{G})| \leq 2|V_{\mathrm{HC}}| \leq 2(c+1)n/(d \log n)$. We then roll the processes in $\mathrm{Pmt}(\overline{G})$ forward (*i.e.*, schedule only these processes, and temporarily pause all other processes, until every process in $\mathrm{Pmt}(\overline{G})$ reaches its noncritical section) by inductively constructing computations $G_0, G_1, \ldots, G_k$ (where $G_0 = \overline{G}$), such that each computation G_{j+1} contains one more critical event (of some process in $\mathrm{Pmt}(\overline{G})$) than G_j. (During the inductive construction, we may erase some active processes in order to eliminate conflicts generated by newly appended events, as explained below.) The computation $\overline{G}$ and the construction of $G_0, \ldots, G_k$ are depicted in Figure 5.7.

If any process p in $\mathrm{Pmt}(\overline{G})$ executes at least $\log n$ critical events before returning to its noncritical section, then the lower bound easily follows. (It can be shown that $j + \log n = \Omega(\log N/ \log\log N)$. The formal argument is presented in Theorem 5.2.) Therefore, we can assume that each process in $\mathrm{Pmt}(\overline{G})$ performs fewer than $\log n$ critical events while being rolled forward. Because $|\mathrm{Pmt}(\overline{G})| \leq 2(c+1)n/(d \log n)$, it follows that all the processes in $\mathrm{Pmt}(\overline{G})$ can be rolled forward with a total of $O(n)$ critical events.

Since each process in $\mathrm{Pmt}(\overline{G})$ is eventually rolled forward and reaches its noncritical section (see Figure 5.7), we do not have to prevent information flow among these processes. (Once all processes in $\mathrm{Pmt}(\overline{G})$ are rolled forward, other processes may freely read variables written by them — knowledge of another process in its noncritical section cannot cause an active process to block.) As before, it can be shown that noncritical events do not generate any information flow that has to be prevented. (In particular, if a noncritical remote read by a process p is appended, then p must have previously

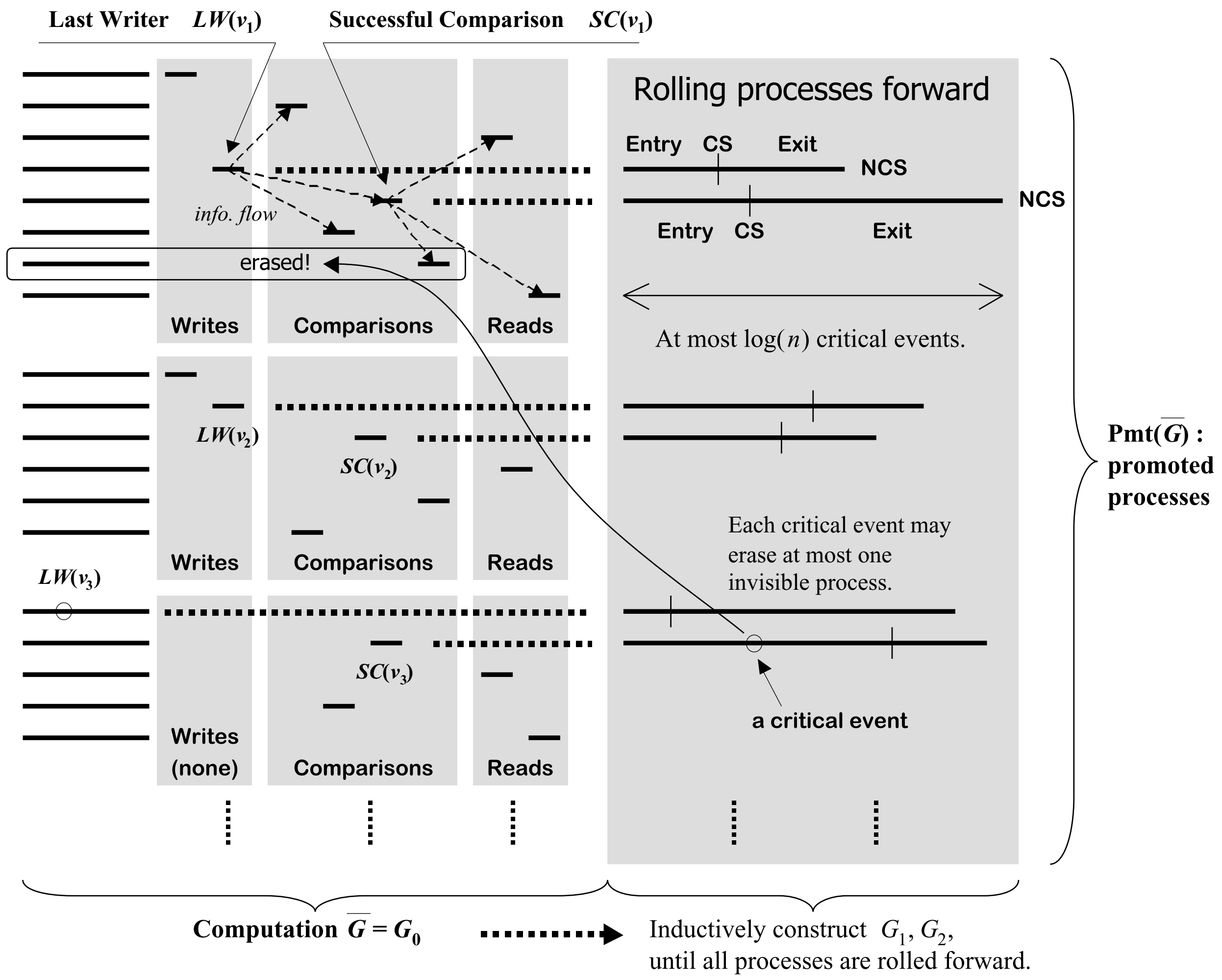

Last Writer $LW(v_1)$
Successful Comparison $SC(v_1)$
info. flow
erased!
Writes
Comparisons
Reads
$LW(v_2)$
$SC(v_2)$
$LW(v_3)$
$SC(v_3)$
Writes (none)
Rolling processes forward
Entry
CS
Exit
NCS
At most log(n) critical events.
Each critical event may erase at most one invisible process.
a critical event
Pmt($\overline{G}$) : promoted processes
Computation $\overline{G} = G_0$
Inductively construct G_1, G_2, until all processes are rolled forward.

Figure 5.7: Roll-forward strategy. Part II.

read the same variable. By Condition RF3, if the last writer is another process, then that process is in the RF-set, and hence is allowed to be known to other processes.) Therefore, the only case to consider is when a critical event of $\mathrm{Pmt}(\overline{G})$ reads a variable v that is written by another active process $q \notin \mathrm{Pmt}(\overline{G})$, *i.e.*, $q \in \mathrm{Inv}(\overline{G})$.

If there are multiple processes in $\mathrm{Act}(\overline{G})$ that write to v in $\overline{G}$, then Condition RF3 guarantees that the last writer in $\overline{G}$ belongs to the RF-set, *i.e.*, we have $q \notin \mathrm{Inv}(\overline{G})$. On the other hand, if there is a single process in $\mathrm{Act}(\overline{G})$ that writes to v in $\overline{G}$, then that process must be q, and information flow can be prevented by erasing q. It follows that each critical event of $\mathrm{Pmt}(\overline{G})$ can conflict with, and thus erase, at most one process in $\mathrm{Inv}(\overline{G})$.

Therefore, the entire roll-forward procedure erases $O(n)$ processes from $\mathrm{Inv}(\overline{G})$. Because $|\mathrm{Inv}(\overline{G})| = \Theta(n) - |\mathrm{Pmt}(\overline{G})|$ and $|\mathrm{Pmt}(\overline{G})| = O(n)$, we can adjust constant coefficients so that $|\mathrm{Pmt}(\overline{G})|$ is at most n multiplied by a small constant (< 1), in which case $\Omega(n)$ processes (*i.e.*, processes in $\mathrm{Inv}(\overline{G})$) survive after the entire procedure. Thus, we can construct H_{j+1}.

5.3 Detailed Lower-bound Proof

In this section, we present our lower-bound theorem. We begin by stating several lemmas. Full proofs for Lemmas 5.1–5.6 can be found in Appendix B. In many of these proofs, computations with a valid RF-set *RFS* are considered. When this is the case, we omit *RFS* when quoting properties RF1–RF5. For example, "H satisfies RF1" means that "H satisfies RF1 when the RF-set under consideration is *RFS*." Throughout this section, as well as in Appendix B, we assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$.

According to Lemma 5.1, stated next, any invisible process can be safely "erased."

Lemma 5.1 Consider a computation H and two sets of processes *RFS* and Y. Assume the following:

- $H \in C$; (5.1)
- *RFS* is a valid RF-set of H; (5.2)
- $RFS \subseteq Y$. (5.3)

Then, the following hold: $H \mid Y \in C$; *RFS* is a valid RF-set of $H \mid Y$; an event e in $H \mid Y$ is a critical event if and only if it is also a critical event in H.

Proof sketch: Because H satisfies RF1, if a process p is not in *RFS*, no process other than p reads a value written by p. Therefore, $H \mid Y \in C$. Conditions RF1–RF5 can be individually checked to hold in $H \mid Y$, which implies that *RFS* is a valid RF-set of $H \mid Y$.

To show that an event e_p in $H \mid Y$ is a critical event if and only if it is also a critical event in H, it is enough to consider critical writes and comparisons. (Transition events and critical reads are straightforward.) In this case, RF5 implies that e_p is critical in $H \mid Y$ if and only if it is also critical in $H \mid \{p\} \cup RFS$, which in turn holds if and only if it is also critical in H. $\square$

The next lemma shows that the property of being a critical event is conserved across "similar" computations. Informally, if process p cannot distinguish two computations H and H', and if p may execute a critical event e_p after H, then it can also execute a critical event e'_p after $H' \circ G$, where G is a computation that does not contain any events by p. Moreover, if G satisfies certain conditions, then $H' \circ G \circ \langle e'_p \rangle$ satisfies RF5, preserving the "criticalness" of e'_p across related computations.

Lemma 5.2 Consider three computations H, H', and G, a set of processes RFS, and two events e_p and e'_p of a process p. Assume the following:

- $H \circ \langle e_p \rangle \in C$; (5.4)
- $H' \circ G \circ \langle e'_p \rangle \in C$; (5.5)
- RFS is a valid RF-set of H; (5.6)
- RFS is a valid RF-set of H'; (5.7)
- $e_p \sim e'_p$; (5.8)
- $p \in \mathrm{Act}(H)$; (5.9)
- $H \mid (\{p\} \cup RFS) = H' \mid (\{p\} \cup RFS)$; (5.10)
- $G \mid p = \langle\rangle$; (5.11)
- no events in G write any of p's local variables; (5.12)
- e_p is critical in $H \circ \langle e_p \rangle$. (5.13)

Then, e'_p is critical in $H' \circ G \circ \langle e'_p \rangle$. Moreover, if the following conditions are true,

(A) $H' \circ G$ satisfies RF5;

(B) if e_p is a comparison event on a variable v, and if G contains a write to v, then $G \mid RFS$ also contains a write to v.

then $H' \circ G \circ \langle e'_p \rangle$ also satisfies RF5

Proof sketch: It is enough to consider the following case: e_p is a critical write or a critical comparison on v such that $writer(v, H) = q$ holds for some process q, where $q \neq \perp$ and $q \neq p$. (As before, if e_p is a transition event or critical read, then the reasoning is straightforward.) If e_p is a critical write, then by applying RF3 to H, we can show $q \in RFS$. Thus, by (5.10), q also writes to v in H' after p's last write to v. Hence, e'_p is critical in $H' \circ G \circ \langle e'_p \rangle$.

Assume that e_p is a critical comparison. If G contains a write to v, then by (5.11), e_p is the first comparison event on v by p after G, and hence is critical by definition. On the other hand, if G does not contain a write to v, then by (5.12), we can show that e_p and e'_p read the same value for each variable they read, and hence by P2 and P5 (given on page 92), we have $e_p = e'_p$. Thus, e'_p is a successful (respectively, unsuccessful) comparison if and only if e_p is also a successful (respectively, unsuccessful) comparison. As in the case of a critical write, by applying RF3 to H, we can show $q \in \mathit{RFS}$. Using this fact, the conditions given in the definition of a critical successful/unsuccessful comparison can be individually checked to hold for e'_p.

If Condition (A) holds, then in order to show that $H' \circ G \circ \langle e'_p \rangle$ satisfies RF5, it suffices to consider e'_p. Condition (B) guarantees that if e'_p is critical because of a write to v in G, then e'_p is also critical in $(H' \circ G \circ \langle e'_p \rangle) \mid (\{p\} \cup \mathit{RFS})$. □

The next lemma provides means of appending an event e_p of an active process, while maintaining RF1 and RF2. This lemma is used inductively in order to extend a computation with a valid RF-set. Specifically, (5.20) guarantees that RF2 is satisfied, and (5.21) forces any information flow to originate from a process in RFS, thus satisfying RF1. (Note that, if $q = \bot$, $q = p$, or $v_{\mathrm{rem}} \notin \mathit{Rvar}(e_p)$ holds, then no information flow occurs.) The proof of this lemma is mainly technical in nature and is omitted here.

Lemma 5.3 Consider two computations H and G, a set of processes RFS, and an event e_p of a process p. Assume the following:

- $H \circ G \circ \langle e_p \rangle \in C$; (5.14)
- RFS is a valid RF-set of H; (5.15)
- $p \in \mathrm{Act}(H)$; (5.16)
- $H \circ G$ satisfies RF1 and RF2; (5.17)
- G is an $\mathrm{Act}(H)$-computation; (5.18)
- $G \mid p = \langle\rangle$; (5.19)
- if e_p remotely accesses a variable v_{rem}, then the following hold:
  - if v_{rem} is local to a process q, then either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq \mathit{RFS}$, and (5.20)
  - if $q = \mathit{writer}(v_{\mathrm{rem}}, H \circ G)$, then one of the following hold: $q = \bot$, $q = p$, $q \in \mathit{RFS}$, or $v_{\mathrm{rem}} \notin \mathit{Rvar}(e_p)$. (5.21)

Then, $H \circ G \circ \langle e_p \rangle$ satisfies RF1 and RF2. □

The next lemma gives us means for extending a computation by appending noncritical events.

Lemma 5.4 Consider a computation H, a set of processes RFS, and another set of processes $Y = \{p_1,\ p_2,\ \ldots,\ p_m\}$. Assume the following:

- $H \in C$; (5.22)
- RFS is a valid RF-set of H; (5.23)
- $Y \subseteq \mathrm{Inv}_{RFS}(H)$; (5.24)
- for each p_j in Y, there exists a computation L_{p_j}, satisfying the following:
  - L_{p_j} is a p_j-computation; (5.25)
  - $H \circ L_{p_j} \in C$; (5.26)
  - L_{p_j} has no critical events in $H \circ L_{p_j}$, that is, no event in L_{p_j} is a critical event in $H \circ L_{p_j}$. (5.27)

Define L to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_m}$. Then, the following hold: $H \circ L \in C$, RFS is a valid RF-set of $H \circ L$, and L contains no critical events in $H \circ L$.

Proof sketch: For each j, define L^j to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_j}$. The lemma can be proved by induction on j. At each induction step, it is assumed that $H \circ L^j \in C$, RFS is a valid RF-set of $H \circ L^j$, and L^j contains no critical events in $H \circ L^j$. Because $L_{p_{j+1}}$ contains no critical events in $H \circ L_{p_{j+1}}$, it can be appended to $H \circ L^j$ to get $H \circ L^{j+1}$ for the next induction step. (As mentioned at the end of Section 5.2, appending a noncritical event cannot cause any undesired information flow from invisible processes to processes in RFS.) □

The next lemma states that if n active processes are competing for entry into their critical sections, then at least $n-1$ of them execute at least one more critical event before entering their critical sections.

Lemma 5.5 Let H be a computation. Assume the following:

- $H \in C$, and (5.28)
- H is regular (i.e., $\mathrm{Fin}(H)$ is a valid RF-set of H). (5.29)

Define $n = |\mathrm{Act}(H)|$. Then, there exists a subset Y of $\mathrm{Act}(H)$, where $n-1 \leq |Y| \leq n$, satisfying the following: for each process p in Y, there exist a p-computation L_p and an event e_p by p such that

- $H \circ L_p \circ \langle e_p \rangle \in C$; (5.30)
- L_p contains no critical events in $H \circ L_p$; (5.31)
- $e_p \notin \{Enter_p, CS_p, Exit_p\}$; (5.32)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (5.33)
- e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$. (5.34)

Proof sketch: First, we construct, for each process p in $\mathrm{Act}(H)$, a computation L_p and an event e_p that satisfy (5.30) and (5.31). Then, we show that every event e_p thus constructed, except at most one, satisfies (5.32). The other conditions can be easily proved and will be omitted here.

Define $H_p = H \mid (\{p\} \cup \mathrm{Fin}(H))$. Because H is regular, $\mathrm{Fin}(H)$ is a valid RF-set of H. Hence, by Lemma 5.1, H_p is in C and $\mathrm{Fin}(H)$ is a valid RF-set of H_p. Since $p \in \mathrm{Act}(H)$, we have $\mathrm{Act}(H_p) = \{p\}$ and $\mathrm{Fin}(H_p) = \mathrm{Fin}(H)$. Therefore, by the Progress property, there exists a p-computation F_p such that $H_p \circ F_p \circ \langle CS_p \rangle \in C$. If F_p has a critical event, then let e'_p be the first critical event in F_p, and let L_p be the prefix of F_p that precedes e'_p (*i.e.*, $F_p = L_p \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L_p to be F_p and e'_p to be CS_p.

Now we have a p-computation L_p and an event e'_p by p, such that $H_p \circ L_p \circ \langle e'_p \rangle \in C$, in which L_p has no critical events and e'_p is a critical event. Because L_p has no critical events in $H_p \circ L_p$, it can be shown that $H \circ L_p \in C$ and that L_p has no critical events in $H \circ L_p$. Because $H \circ L_p$ and $H_p \circ L_p$ are equivalent with respect to p, by P3, there exists an event e_p by p such that $e_p \sim e'_p$ and $H \circ L_p \circ \langle e_p \rangle \in C$.

We now claim that at most one process in $\mathrm{Act}(H)$ fails to satisfy (5.32). Because $p \in \mathrm{Act}(H)$ and H is regular, e_p cannot be $Enter_p$ or $Exit_p$. By the Exclusion property, there can be at most one $p \in \mathrm{Act}(H)$ such that $e_p = CS_p$. $\square$

The following lemma is used to roll processes forward. It states that as long as there exist promoted processes, we can extend the computation with one more critical event of some promoted process, and at most one invisible process must be erased due to the resulting information flow.

Lemma 5.6 Consider a computation H and set of processes RFS. Assume the following:

- $H \in C$; (5.35)
- RFS is a valid RF-set of H; (5.36)
- $\mathrm{Fin}(H) \subsetneq RFS$ (*i.e.*, $\mathrm{Fin}(H)$ is a proper subset of RFS). (5.37)

Then, there exists a computation G satisfying the following.

- $G \in C$; (5.38)
- RFS is a valid RF-set of G; (5.39)
- G can be written as $H \mid (Y \cup RFS) \circ L \circ \langle e_p \rangle$, for some choice of Y, L, and e_p, satisfying the following:
  - Y is a subset of $\mathrm{Inv}(H)$ such that $|\mathrm{Inv}(H)| - 1 \le |Y| \le |\mathrm{Inv}(H)|$, (5.40)

  - $\text{Inv}(G) = Y$, (5.41)
  - L is a $\text{Pmt}(H)$-computation, (5.42)
  - L has no critical events in G, (5.43)
  - $p \in \text{Pmt}(H)$, and (5.44)
  - e_p is critical in G; (5.45)
- $\text{Pmt}(G) \subseteq \text{Pmt}(H)$; (5.46)
- An event in $H \mid (Y \cup RFS)$ is critical if and only if it is also critical in H. (5.47)

Proof sketch: Let $H' = H \mid RFS$ and $Z = \text{Pmt}(H)$. Then, by the definition of an active process, $\text{Act}(H') = (\text{Act}(H) \cap RFS) = \text{Pmt}(H) = Z$. By Lemma 5.1, $H' \in C$ and RFS is a valid RF-set of H'.

Therefore, by applying the Progress property, we can construct a Z-computation F such that $H' \circ F \circ \langle \bar{e}_r \rangle \in C$, where r is a process in Z and $\bar{e}_r$ is either CS_r or $Exit_r$. If F has a critical event, then let e'_p be the first critical event in F, and let L be the prefix of F that precedes e'_p (*i.e.*, $F = L \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L to be F and e'_p to be $\bar{e}_r$. Because F is a Z-computation, $p \in Z$.

Now we have a Z-computation L and an event e'_p by $p \in Z$, such that $H' \circ L \circ \langle e'_p \rangle \in C$, L has no critical events in $H' \circ L \circ \langle e'_p \rangle$, and e'_p is a critical event in $H' \circ L \circ \langle e'_p \rangle$. It can be shown that $H \circ L \in C$ and that L has no critical events in $H \circ L$. (This follows because H and H' are equivalent with respect to Z.) Because $H \circ L$ and $H' \circ L$ are equivalent with respect to p, by P3, there exists an event e''_p by p such that $e''_p \sim e'_p$ and $H \circ L \circ \langle e''_p \rangle \in C$.

Because e'_p is a critical event in $H' \circ L \circ \langle e'_p \rangle$ and e''_p accesses the same variables as e'_p, it can be shown that e''_p is a critical event in $H \circ L \circ \langle e''_p \rangle$. Let v be the remote variable accessed by e''_p. If v is local to a process q in $\text{Inv}(H)$, or if $q = \mathit{writer}(v, H \circ L)$ is in $\text{Inv}(H)$, then we can "erase" process q and construct a computation G that satisfies the requirements stated in the lemma. (If both conditions hold simultaneously, then by RF2, q is identical in both cases.) The event e_p that is appended to obtain G is congruent to e''_p, *i.e.*, $e_p \sim e''_p$. □

The following theorem is due to Turán [81].

Theorem 5.1 (Turán) *Let $\mathcal{G} = (V, E)$ be an undirected graph, with vertex set V and edge set E. If the average degree of $\mathcal{G}$ is d, then an independent set*[12] *exists with at least $\lceil |V|/(d+1) \rceil$ vertices.* □

The following lemma provides the induction step that leads to the lower bound in Theorem 5.2.

Lemma 5.7 Let H be a computation. Assume the following:

- $H \in C$, and (5.48)
- H is regular (*i.e.*, $\mathrm{Fin}(H)$ is a valid RF-set of H). (5.49)

Define $n = |\mathrm{Act}(H)|$. Also assume that

- $n > 1$, and (5.50)
- each process in $\mathrm{Act}(H)$ executes exactly c critical events in H, where $c \le \log n - 1$. (5.51)

Then, one of the following propositions is true.

Pr1: There exist a process p in $\mathrm{Act}(H)$ and a computation F in C such that

- $F \circ \langle \mathit{Exit}_p \rangle \in C$;
- F does not contain $\langle \mathit{Exit}_p \rangle$;
- p executes at least $(c + \log n)$ critical events in F.

Pr2: There exists a regular computation G in C such that

- $\mathrm{Act}(G) \subseteq \mathrm{Act}(H)$; (5.52)
- $|\mathrm{Act}(G)| \ge \min\left(\dfrac{n}{6} - \dfrac{n}{2\log n} - \dfrac{1}{2}, \dfrac{n-1}{2 \cdot (12\log^2 n + 1)}\right)$; (5.53)
- each process in $\mathrm{Act}(G)$ executes exactly $(c+1)$ critical events in G. (5.54)

Proof: We first apply Lemma 5.5. Assumptions (5.28) and (5.29) stated in Lemma 5.5 follow from (5.48) and (5.49), respectively. It follows that there exists a set of processes Y such that

- $Y \subseteq \mathrm{Act}(H)$, and (5.55)
- $n - 1 \le |Y| \le n$, (5.56)

and for each process $p \in Y$, there exist a computation L_p and an event e_p by p, such that

- $H \circ L_p \circ \langle e_p \rangle \in C$; (5.57)
- L_p is a p-computation; (5.58)

[12]An *independent set* of a graph $\mathcal{G} = (V, E)$ is a subset $V' \subseteq V$ such that no edge in E is incident to two vertices in V'.

- L_p contains no critical events in $H \circ L_p$; (5.59)
- $e_p \notin \{\mathit{Enter}_p, \mathit{CS}_p, \mathit{Exit}_p\}$; (5.60)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (5.61)
- e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$. (5.62)

For each $p \in Y$, by (5.58), (5.59), and $p \in Y \subseteq \mathrm{Act}(H)$, we have

$$\mathrm{Act}(H \circ L_p) = \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H \circ L_p) = \mathrm{Fin}(H). \tag{5.63}$$

By (5.50) and (5.56), Y is nonempty.

If Proposition Pr1 is satisfied by any process in Y, then the theorem is clearly true. Thus, we will assume, throughout the remainder of the proof, that there is no process in Y that satisfies Pr1. Define $\mathcal{E}_H$ as the set of critical events in H of processes in Y.

$$\mathcal{E}_H = \{f_q \text{ in } H\colon f_q \text{ is critical in } H \text{ and } q \in Y\}. \tag{5.64}$$

Define $\mathcal{E} = \mathcal{E}_H \cup \{e_p\colon p \in Y\}$, *i.e.*, the set of all "past" and "next" critical events of processes in Y. From (5.51), (5.55), and (5.56), it follows that

$$|\mathcal{E}| = (c+1)|Y| \leq (c+1)n. \tag{5.65}$$

Now define V_{HC}, the set of variables that experience "high contention" (*i.e.*, those that are accessed by "sufficiently many" events in $\mathcal{E}$), as follows.

$$V_{\mathrm{HC}} = \{v \in V\colon \text{there are at least } 6\log^2 n \text{ events in } \mathcal{E} \text{ that remotely access } v\}. \tag{5.66}$$

Since, by the Atomicity property (given on page 90), each event in $\mathcal{E}$ can access at most one remote variable, from (5.51) and (5.65), we have

$$|V_{\mathrm{HC}}| \leq \frac{|\mathcal{E}|}{6\log^2 n} \leq \frac{(c+1)n}{6\log^2 n} \leq \frac{n}{6\log n}. \tag{5.67}$$

Define P_{HC}, the set of processes whose "next" event accesses a variable in V_{HC}, as follows.

$$P_{\mathrm{HC}} = \{p \in Y\colon e_p \text{ accesses a variable in } V_{\mathrm{HC}}\}. \tag{5.68}$$

We now consider two cases, depending on $|P_{\mathrm{HC}}|$.

Case 1: $|P_{\mathrm{HC}}| < \frac{1}{2}|Y|$ (erasing strategy)
— *In this case, we start with $Y' = Y - P_{\mathrm{HC}}$, which consists of at least $(n-1)/2$ active processes. We construct a "conflict graph" $\mathcal{G}$, made of the processes in Y'. By applying Theorem 5.1, we can find a subset Z of Y' such that their critical events do not conflict with each other.*

Let $Y' = Y - P_{\mathrm{HC}}$. By (5.55), we have

$$Y' \subseteq \mathrm{Act}(H). \tag{5.69}$$

By (5.56) and Case 1, we also have

$$|Y'| \;=\; \big(|Y| - |P_{\mathrm{HC}}|\big) \;>\; \left(|Y| - \frac{1}{2}Y\right) \;=\; \frac{1}{2}|Y| \;\geq\; \frac{n-1}{2}. \tag{5.70}$$

We now construct an undirected graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y'. To each process y in Y' and each variable $v \in \mathit{var}(e_y)$ that is remote to y, we apply the following rules.

- **R1:** If v is local to a process z in Y', then introduce edge $\{y, z\}$.
- **R2:** If there exists an event $f_p \in \mathcal{E}$ that remotely accesses v, and if $p \in Y'$, then introduce edge $\{y, p\}$.

Because each variable is local to at most one process, and since (by the Atomicity property, given on page 90) an event can access at most one remote variable, Rule R1 can introduce at most one edge for each process in Y. Since $y \in Y'$, we have $y \notin P_{\mathrm{HC}}$, which, by (5.68), implies $v \notin V_{\mathrm{HC}}$. Hence, by (5.66), it follows that there are at most $6\log^2 n - 1$ events in $\mathcal{E}$ that remotely access v. Therefore, since an event can access at most one remote variable, Rule R2 can introduce at most $6\log^2 n - 1$ edges for each process in Y.

Combining Rules R1 and R2, at most $6\log^2 n$ edges are introduced for each process in Y. Since each edge is counted twice (for each of its endpoints), the average degree of $\mathcal{G}$ is at most $12\log^2 n$. Hence, by Theorem 5.1, there exists an independent set Z such that

$$Z \subseteq Y', \quad \text{and} \tag{5.71}$$

$$|Z| \geq \frac{|Y'|}{(12\log^2 n + 1)} \geq \frac{n-1}{2\cdot(12\log^2 n + 1)}, \tag{5.72}$$

where the latter inequality follows from (5.70).

Next, we construct a computation G, satisfying Proposition Pr2, such that $\mathrm{Act}(G) = |Z|$.

Define H' as

$$H' = H \mid (Z \cup \mathrm{Fin}(H)). \tag{5.73}$$

By (5.69) and (5.71), we have

$$Z \subseteq Y' \subseteq Y \subseteq \mathrm{Act}(H), \tag{5.74}$$

and hence,

$$\mathrm{Act}(H') = Z \subseteq \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{5.75}$$

We now apply Lemma 5.1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$ and 'Y' $\leftarrow Z \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (5.48) and (5.49), respectively; (5.3) is trivial. It follows that

- $H' \in C$, (5.76)
- $\mathrm{Fin}(H)$ is a valid RF-set of H', and (5.77)
- an event in H' is critical if and only if it is also critical in H. (5.78)

Our goal now is to show that H' can be extended so that each process in Z has one more critical event. By (5.75), (5.77), and by the definition of a finished process,

$$\mathrm{Inv}_{\mathrm{Fin}(H)}(H') = \mathrm{Act}(H') = Z. \tag{5.79}$$

For each $z \in Z$, define F_z as

$$F_z = (H \circ L_z) \mid (Z \cup \mathrm{Fin}(H)). \tag{5.80}$$

By (5.74), we have $z \in Y$. Thus, applying (5.57), (5.58), (5.59), and (5.61) with 'p' $\leftarrow z$, it follows that

- $H \circ L_z \circ \langle e_z \rangle \in C$; (5.81)
- L_z is a z-computation; (5.82)
- L_z contains no critical events in $H \circ L_z$; (5.83)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_z$. (5.84)

By P1, (5.81) implies

$$H \circ L_z \in C. \tag{5.85}$$

We now apply Lemma 5.1, with 'H' $\leftarrow H \circ L_z$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow Z \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (5.85) and (5.84), respectively; (5.3) is trivial. It follows that

- $F_z \in C$, and (5.86)
- an event in F_z is critical if and only if it is also critical in $H \circ L_z$. (5.87)

Since $z \in Z$, by (5.73), (5.80), and (5.82), we have

$$F_z = H' \circ L_z.$$

Hence, by (5.83) and (5.87),

- L_z contains no critical events in $F_z = H' \circ L_z$. (5.88)

Let $m = |Z|$ and index the processes in Z as $Z = \{z_1,\ z_2,\ \ldots,\ z_m\}$. Define $L = L_{z_1} \circ L_{z_2} \circ \cdots \circ L_{z_m}$. We now use Lemma 5.4, with 'H' $\leftarrow H'$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow Z$, and 'p_j' $\leftarrow z_j$ for each $j = 1, \ldots, m$. Among the assumptions stated in Lemma 5.4, (5.22)–(5.24) follow from (5.76), (5.77), and (5.79), respectively; (5.25)–(5.27) follow from (5.82), (5.86), and (5.88), respectively, with 'z' $\leftarrow z_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (5.89)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (5.90)
- L contains no critical events in $H' \circ L$. (5.91)

To this point, we have successfully appended a (possibly empty) sequence of non-critical events for each process in Z. It remains to append a "next" critical event for each such process. Note that, by (5.82) and the definition of L,

- L is a Z-computation. (5.92)

Thus, by (5.75) and (5.91), we have

$$\mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = Z \quad \wedge \quad \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{5.93}$$

By (5.73) and the definition of L, it follows that

- for each $z \in Z$, $(H \circ L_z) \mid (\{z\} \cup \mathrm{Fin}(H)) = (H' \circ L) \mid (\{z\} \cup \mathrm{Fin}(H))$. (5.94)

In particular, $H \circ L_z$ and $H' \circ L$ are equivalent with respect to z. Therefore, by (5.81), (5.89), and repeatedly applying P3, it follows that, for each $z_j \in Z$, there exists an event e'_{z_j}, such that

- $G \in C$, where $G = H' \circ L \circ E$ and $E = \langle e'_{z_1},\ e'_{z_2},\ \ldots,\ e'_{z_m}\rangle$; (5.95)
- $e'_{z_j} \sim e_{z_j}$. (5.96)

By the definition of E,

- E is a Z-computation. (5.97)

By (5.60), (5.93), and (5.96), we have

$$\mathrm{Act}(G) = \mathrm{Act}(H' \circ L) = Z \quad \wedge \quad \mathrm{Fin}(G) = \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H). \tag{5.98}$$

By (5.60), (5.62), and (5.96), it follows that for each $z_j \in Z$, both e_{z_j} and e'_{z_j} access a common remote variable, say, v_j. Since Z is an independent set of $\mathcal{G}$, by Rules R1 and R2, we have the following:

- for each $z_j \in Z$, v_j is not local to any process in Z; (5.99)
- $v_j \neq v_k$, if $j \neq k$.

Combining these two, we also have:

- for each $z_j \in Z$, no event in E other than e'_{z_j} accesses v_j (either locally or remotely). (5.100)

We now establish two claims.

Claim 1: For each $z_j \in Z$, if we let $q = \mathit{writer}(v_j, H' \circ L)$, then one of the following holds: $q = \perp$, $q = z_j$, or $q \in \mathrm{Fin}(H)$.

Proof of Claim: It suffices to consider the case when $q \neq \perp$ and $q \neq z_j$ hold, in which case there exists an event f_q by q in $H' \circ L$ that writes to v_j. By (5.73) and (5.92), we have $q \in Z \cup \mathrm{Fin}(H)$. We claim that $q \in \mathrm{Fin}(H)$ holds in this case. Assume, to the contrary,

$$q \in Z. \tag{5.101}$$

We consider two cases. First, if f_q is a critical event in $H' \circ L$, then by (5.91), f_q is an event of H', and hence, by (5.78), f_q is also a critical event in H. By (5.74) and (5.101), we have $q \in Y$. Thus, by (5.64), we have $f_q \in \mathcal{E}_H$, and hence $f_q \in \mathcal{E}$ holds by definition. By (5.99) and (5.101), v_j is remote to q. Thus, f_q *remotely* writes v_j. By (5.101) and $z_j \in Z$, we have

$$\{q, z_j\} \subseteq Z, \tag{5.102}$$

which implies $\{q, z_j\} \subseteq Y'$ by (5.71). From this, our assumption of $q \neq z_j$, and by applying Rule R2 with 'y' $\leftarrow z_j$ and 'f_p' $\leftarrow f_q$, it follows that edge $\{q, z_j\}$ exists in $\mathcal{G}$. However, (5.102) then implies that Z is not an independent set of $\mathcal{G}$, a contradiction.

Second, assume that f_q is a noncritical event in $H' \circ L$. Note that, by (5.99) and (5.101), v_j is remote to q. Hence, by the definition of a critical event, there exists a critical event $\bar{f}_q$ by q in $H' \circ L$ that remotely writes to v_j. However, this leads to contradiction as shown above. □

Claim 2: Every event in E is critical in G. Also, G satisfies RF5 with 'RFS' $\leftarrow \text{Fin}(H)$.

Proof of Claim: Define $E_0 = \langle\rangle$; for each positive j, define E_j to be $\langle e'_{z_1}, e'_{z_2}, \ldots, e'_{z_j}\rangle$, a prefix of E. We prove the claim by induction on j, applying Lemma 5.2 at each step. Note that, by (5.95) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{z_{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{5.103}$$

Also, by the definition of E_j, we have

$$E_j \mid z_{j+1} = \langle\rangle, \quad \text{for each } j. \tag{5.104}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5 with 'RFS' $\leftarrow \text{Fin}(H)$. (5.105)

The induction base ($j = 0$) follows easily from (5.90), since $E_0 = \langle\rangle$.

Assume that (5.105) holds for a particular value of j. Since $z_{j+1} \in Z$, by (5.74), we have

$$z_{j+1} \in Y, \tag{5.106}$$

and $z_{j+1} \in \mathrm{Act}(H)$. By applying (5.63) with 'p' $\leftarrow z_{j+1}$, and using (5.106), we also have $\mathrm{Act}(H \circ L_{z_{j+1}}) = \mathrm{Act}(H)$, and hence

$$z_{j+1} \in \mathrm{Act}(H \circ L_{z_{j+1}}). \tag{5.107}$$

By (5.104), if any event e'_{z_k} in E_j accesses a local variable v of z_{j+1}, then e'_{z_k} accesses v *remotely*, and hence $v = v_k$ by definition. However, by (5.99), v_k cannot be local to z_{j+1}. It follows that

- no events in E_j access any of z_{j+1}'s local variables. (5.108)

We now apply Lemma 5.2, with 'H' $\leftarrow H \circ L_{z_{j+1}}$, '$H'$' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e_{z_{j+1}}$, and 'e'_p' $\leftarrow e'_{z_{j+1}}$. Among the assumptions stated in Lemma 5.2, (5.5), (5.7), (5.9), (5.11), and (5.12) follow from (5.103), (5.90), (5.107), (5.104), and (5.108), respectively; (5.8) follows by applying (5.96) with 'z_j' $\leftarrow z_{j+1}$; (5.6) and (5.10) follow by applying (5.84) and (5.94), respectively, with 'z' $\leftarrow z_{j+1}$; and (5.4) and (5.13) follow by applying (5.57) and (5.62), respectively, with 'p' $\leftarrow z_{j+1}$, and using (5.106). Moreover, Assumption (A) follows from (5.105), and Assumption (B) is satisfied vacuously (with 'v' $\leftarrow v_{j+1}$) by (5.100).

It follows that $e'_{z_{j+1}}$ is critical in $H' \circ L \circ E_j \circ \langle e'_{z_{j+1}} \rangle = H' \circ L \circ E_{j+1}$, and that $H' \circ L \circ E_{j+1}$ satisfies RF5 with 'RFS' $\leftarrow \mathrm{Fin}(H)$. □

We now claim that $\mathrm{Fin}(H)$ is a valid RF-set of G. Condition RF5 was already proved in Claim 2.

- **RF1 and RF2:** Define E_j as in Claim 2. We establish RF1 and RF2 by induction on j, applying Lemma 5.3 at each step. At each step, we assume

  - $H' \circ L \circ E_j$ satisfies RF1 and RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$. (5.109)

  The induction base ($j = 0$) follows easily from (5.90), since $E_0 = \langle\rangle$.

  Assume that (5.109) holds for a particular value of j. Note that, by (5.100), we have $\mathit{writer}(v_{j+1}, H' \circ L \circ E_j) = \mathit{writer}(v_{j+1}, H' \circ L)$. Thus, by (5.93) and Claim 1,

  - if we let $q = \mathit{writer}(v_{j+1}, H' \circ L \circ E_j)$, then one of the following holds: $q = \perp$, $q = z_{j+1}$, or $q \in \mathrm{Fin}(H) = \mathrm{Fin}(H' \circ L)$. (5.110)

We now apply Lemma 5.3, with 'H' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e'_{z_{j+1}}$, and 'v_{rem}' $\leftarrow v_{j+1}$. Among the assumptions stated in Lemma 5.3, (5.14), (5.15), (5.17), (5.19), and (5.21) follow from (5.103), (5.90), (5.109), (5.104), and (5.110), respectively; (5.16) follows from (5.93) and $z_{j+1} \in Z$; (5.18) follows from (5.93) and (5.97); (5.20) follows from (5.99) and (5.93). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$.

- **RF3:** Consider a variable $v \in V$ and two different events f_q and g_r in G. Assume that both q and r are in $\mathrm{Act}(G)$, $q \neq r$, and that there exists a variable v such that $v \in var(f_q) \cap var(g_r)$. (Note that, by (5.98), $\{q, r\} \subseteq Z$.) We claim that these conditions can actually never arise simultaneously, which implies that G vacuously satisfies RF3.

  Since v is remote to at least one of q or r, without loss of generality, assume that v is remote to q. We claim that there exists an event $\bar{f}_q$ in $\mathcal{E}$ that accesses the same variable v. If f_q is an event of E, we have $f_q = e'_{z_j}$ for some $z_j \in Z$, and $e_{z_j} \in \mathcal{E}$ holds by definition; define $\bar{f}_q = e_{z_j}$ in this case. If f_q is a noncritical event in $H' \circ L$, then by definition of a critical event, there exists a critical event $\bar{f}_q$ in $H' \circ L$ that remotely accesses v. If f_q is a critical event in $H' \circ L$, then define $\bar{f}_q = f_q$. (Note that, if $\bar{f}_q$ is a critical event in $H' \circ L$, then by (5.78) and (5.91), $\bar{f}_q$ is also a critical event in H, and hence, by $q \in Z$, (5.74), and the definition of $\mathcal{E}$, we have $\bar{f}_q \in \mathcal{E}$.)

  It follows that, in each case, there exists an event $\bar{f}_q \in \mathcal{E}$ that remotely accesses v. If v is local to r, then by Rule R1, $\mathcal{G}$ contains the edge $\{q, r\}$. On the other hand, if v is remote to r, then we can choose an event $\bar{g}_r \in \mathcal{E}$ that remotely accesses v, in the same way as shown above. Hence, by Rule R2, $\mathcal{G}$ contains the edge $\{q, r\}$. Thus, in either case, p and q cannot simultaneously belong to Z, a contradiction.

- **RF4:** By (5.90) and (5.98), it easily follows that G satisfies RF4 with respect to $\mathrm{Fin}(H)$.

Finally, we claim that G satisfies Proposition Pr2. By (5.98), we have $\mathrm{Act}(G) = Z \subseteq \mathrm{Act}(H)$, so G satisfies (5.52). By (5.72), we have (5.53). By (5.51), (5.78), and (5.91), each process in Z executes exactly c critical events in $H' \circ L$. Thus, by Claim 2, G satisfies (5.54).

Case 2: $|P_{\mathrm{HC}}| \geq \frac{1}{2}|Y|$ (roll-forward strategy)
— *In this case, we start with* P_{HC}, *which, by* (5.56), *consists of at least* $(n-1)/2$

active processes. We first erase the processes in K, defined below, to satisfy RF2. Appending the critical events e_p for each p in $S = P_{\mathrm{HC}} - K$ gives us a non-regular computation $\overline{G}$. We then select a subset of P_{HC}, which consists of $LW(v)$ and $SC(v)$ for each $v \in V_{\mathrm{HC}}$, as the set of promoted processes. We roll these processes forward, inductively generating a sequence of computations $G_0,\ G_1,\ \ldots,\ G_k$ (where $G_0 = \overline{G}$), where the last computation G_k is regular. We erase at most $n/3$ processes during the procedure, which leaves $\Theta(n)$ active processes in G_k.

Define K, the erased (or "killed") processes, S, the "survivors," and H', the resulting computation, as follows.

$$\begin{aligned} K &= \{p \in P_{\mathrm{HC}}\colon \text{there exists a variable } v \in V_{\mathrm{HC}} \\ &\qquad\qquad \text{such that } v \text{ is local to } p\} && (5.111)\\ S &= P_{\mathrm{HC}} - K && (5.112)\\ H' &= H \mid (S \cup \mathrm{Fin}(H)) && (5.113)\end{aligned}$$

Because each variable is local to at most one process, from (5.67) and (5.111), we have

$$|K| \le \frac{n}{6\log n}. \qquad (5.114)$$

By (5.55), (5.68) and (5.112), we have

$$S \subseteq P_{\mathrm{HC}} \subseteq Y \subseteq \mathrm{Act}(H), \qquad (5.115)$$

and hence,

$$\mathrm{Act}(H') = S \subseteq \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \qquad (5.116)$$

We now apply Lemma 5.1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$ and 'Y' $\leftarrow S \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (5.48) and (5.49), respectively; (5.3) is trivial. It follows that

- $H' \in C$, (5.117)
- $\mathrm{Fin}(H)$ is a valid RF-set of H', and (5.118)
- an event in H' is critical if and only if it is also critical in H. (5.119)

Our goal now is to show that H' can be extended to a computation $\overline{G}$ (defined later in Figure 5.8), so that each process in S has one more critical event. By (5.116), (5.118), and by the definition of a finished process,

$$\mathrm{Inv}_{\mathrm{Fin}(H)}(H') = \mathrm{Act}(H') = S. \tag{5.120}$$

For each $s \in S$, define F_s as

$$F_s = (H \circ L_s) \mid (S \cup \mathrm{Fin}(H)). \tag{5.121}$$

By (5.115), we have $s \in Y$. Thus, applying (5.57), (5.58), (5.59), and (5.61) with 'p' $\leftarrow s$, it follows that

- $H \circ L_s \circ \langle e_s \rangle \in C$; (5.122)
- L_s is an s-computation; (5.123)
- L_s contains no critical events in $H \circ L_s$; (5.124)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_s$. (5.125)

By P1, (5.122) implies

$$H \circ L_s \in C. \tag{5.126}$$

We now apply Lemma 5.1, with 'H' $\leftarrow H \circ L_s$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow S \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (5.126) and (5.125), respectively; (5.3) is trivial. It follows that

- $F_s \in C$, and (5.127)
- an event in F_s is critical if and only if it is also critical in $H \circ L_s$. (5.128)

Since $s \in S$, by (5.113), (5.121), (5.123), and (5.127), we have

- $F_s = H' \circ L_s \in C$. (5.129)

Hence, by (5.124) and (5.128),

- L_s contains no critical events in $F_s = H' \circ L_s$. (5.130)

By (5.111) and (5.112), no variable in V_{HC} is local to any process in S. Therefore, by (5.68) and (5.112),

- for each process s in S, e_s *remotely* accesses a variable in V_{HC}. (5.131)

We now show that the events in $\{L_s\colon s \in S\}$ can be "merged" by applying Lemma 5.4. We arbitrarily index S as $\{s_1, s_2, \ldots, s_m\}$, where $m = |S|$. (Later, we construct a specific indexing of S to reduce information flow.) Let $L = L_{s_1} \circ L_{s_2} \circ \cdots \circ L_{s_m}$. Apply Lemma 5.4, with 'H' $\leftarrow H'$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow S$, and 'p_j' $\leftarrow s_j$ for each

$j = 1, \ldots, m$. Among the assumptions stated in Lemma 5.4, (5.22)–(5.24) follow from (5.117), (5.118), and (5.120), respectively; (5.25)–(5.27) follow from (5.123), (5.129), and (5.130), respectively, with 's' $\leftarrow s_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (5.132)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (5.133)
- L contains no critical events in $H' \circ L$. (5.134)

By (5.113) and the definition of L, we also have,

- for each $s \in S$, $(H \circ L_s) \mid (\{s\} \cup \mathrm{Fin}(H)) = (H' \circ L) \mid (\{s\} \cup \mathrm{Fin}(H))$; (5.135)
- for each $s \in S$, $(H' \circ L) \mid s = (H \circ L_s) \mid s$. (5.136)

We now re-index the processes in S so that information flow among them is minimized. The re-indexing method is expressed as an algorithm in Figure 5.8. This algorithm is described in Claim 3 below. (The event ordering produced by the algorithm was illustrated earlier in Figure 5.6.) The algorithm constructs an indexing (s^1, s^2, ..., s^m) of S and two computations $\overline{G}$ and E, such that

- $\overline{G} \in C$, where $\overline{G} = H' \circ L \circ E$ and $E = \langle e'_{s^1}, e'_{s^2}, \ldots, e'_{s^m} \rangle$; (5.137)
- $e'_{s^j} \sim e_{s^j}$. (5.138)

By the definition of E,

- E is an S-computation. (5.139)

By (5.60), (5.134), and (5.138), $L \circ E$ does not contain any transition events. Moreover, by the definition of L and E, $(L \circ E) \mid p \neq \langle\rangle$ implies $p \in S$, for each process p. Combining these assertions with (5.116), we have

$$\begin{array}{rcl} \mathrm{Act}(\overline{G}) & = & \mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = S \quad \wedge \\ \mathrm{Fin}(\overline{G}) & = & \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \end{array} \tag{5.140}$$

We now state and prove two claims regarding $\overline{G}$. Claim 3 follows easily by examining the algorithm.

Claim 3: For each $v \in V_{\mathrm{HC}}$, the algorithm in Figure 5.8 constructs four (possibly empty) sets of events, $W(v)$, $C_1(v)$, $C_2(v)$, and $R(v)$, and a value, $\alpha(v)$. All events in E that access v appear contiguously, in the following order.

```
begin
    $j := 0$;    $E_0 := \langle\rangle$;
    for each $v \in V_{\mathrm{HC}}$ do
        $W(v) := \{\}$;    $C_1(v) := \{\}$;    $C_2(v) := \{\}$;    $R(v) := \{\}$
    od;
    for each $v \in V_{\mathrm{HC}}$ do
        for each $s \in S$ such that $op(e_s) = \mathsf{write}(v)$ do
            $append(s)$;    add $e'_s$ to $W(v)$
        od;
        $\alpha(v) := value(v, H' \circ L \circ E_j)$;
        for each $s \in S$ such that $op(e_s) = \mathsf{compare}(v, \beta)$ for any $\beta \neq \alpha(v)$ do
            $append(s)$;    add $e'_s$ to $C_1(v)$
        od;
        for each $s \in S$ such that $op(e_s) = \mathsf{compare}(v, \alpha(v))$ do
            $append(s)$;    add $e'_s$ to $C_2(v)$
        od;
        for each $s \in S$ such that $op(e_s) = \mathsf{read}(v)$ do
            $append(s)$;    add $e'_s$ to $R(v)$
        od
    od;
    $\underline{E} := E_m$;
    $\overline{G} = H' \circ L \circ E$
end

procedure $append(s$ : a process$)$
    INVARIANT: $0 \leq j < m = |S|$;
               $E_j = \langle e'_{s^1}, e'_{s^2}, \ldots, e'_{s^j} \rangle$;
               $(s^1, s^2, \ldots, s^j)$ is a sequence of distinct processes in $S$;
               $H' \circ L \circ E_j \in C$;
               $e'_{s^k} \sim e_{s^k}$ and $s \neq s^k$ for each $k = 1, 2, \ldots, j$.
    $s^{j+1} := s$;
        — By (5.122), (5.132), (5.136), and Property P3, there exists an
          event $e'_{s^{j+1}}$ such that $H' \circ L \circ E_j \circ \langle e'_{s^{j+1}} \rangle \in C$ and $e'_{s^{j+1}} \sim e_{s^{j+1}}$
          hold.
    $E_{j+1} := E_j \circ \langle e'_{s^{j+1}} \rangle$;
    $j := j + 1$
end
```

Figure 5.8: Algorithm for arranging the events of S so that information flow is sufficiently low.

- events in $W(v)$: each event e'_s in $W(v)$ satisfies $op(e'_s) = \mathsf{write}(v)$;
- events in $C_1(v)$: each event e'_s in $C_1(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \beta_s)$ for some $\beta_s \neq \alpha(v)$;
- events in $C_2(v)$: each event e'_s in $C_2(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \alpha(v))$;
- events in $R(v)$: each event e'_s in $R(v)$ satisfies $op(e'_s) = \mathsf{read}(v)$.

Moreover, in the computation $\overline{G}$, after all events in $W(v)$ are executed, and before any event in $C_2(v)$ is executed, v has the value $\alpha(v)$. All events in $C_1(v)$ (if any) are unsuccessful comparisons. At most one event in $C_2(v)$ is a successful comparison. (Note that a successful comparison event writes a value other than $\alpha(v)$, by definition. Thus, if there is a successful comparison, then all subsequent comparison events must fail.) For each $v \in V_{\mathrm{HC}}$, define $LW(v)$, the "last write," and $SC(v)$, the "successful comparison," as follows:

$$LW(v) = \begin{cases} \text{the last event in } W(v), & \text{if } W(v) \neq \{\}, \\ writer\_event(v, H' \circ L), & \text{if } W(v) = \{\}; \end{cases}$$

$$SC(v) = \begin{cases} \text{the successful comparison in } C_2(v), & \text{if } C_2(v) \text{ contains one,} \\ \perp, & \text{otherwise.} \end{cases}$$

Then, the last process to write to v (if any) is either $SC(v)$ (if $SC(v)$ is defined) or $LW(v)$ (otherwise). $\Box$

Before establishing our next claim, Claim 4, we define RFS as

$$\begin{aligned} RFS = {} & \mathrm{Fin}(H) \\ & \cup \{owner(LW(v)) \colon v \in V_{\mathrm{HC}} \text{ and } LW(v) \neq \perp\} \\ & \cup \{owner(SC(v)) \colon v \in V_{\mathrm{HC}} \text{ and } SC(v) \neq \perp\}. \end{aligned} \tag{5.141}$$

By (5.68), (5.111), (5.112), and (5.138), we have the following:

- for each $s \in S$, if e'_s remotely accesses v, and if v is local to a process q, then $q \notin S$. (5.142)

Note that "expanding" a valid RF-set does not falsify any of RF1–RF5. Therefore, using (5.133), (5.140), and $\mathrm{Fin}(H) \subseteq RFS \subseteq \mathrm{Fin}(H) \cup S$, it follows that

- RFS is a valid RF-set of $H' \circ L$. (5.143)

We now establish Claim 4, stated below.

Claim 4: Every event in E is critical in $\overline{G}$. Also, $\overline{G}$ satisfies RF5.

Proof of Claim: Define $E_0 = \langle\rangle$; for each positive j, define E_j to be $\langle e'_{s^1},\ e'_{s^2}, \ldots,\ e'_{s^j}\rangle$, a prefix of E. We prove the claim by induction on j, applying Lemma 5.2 at each step. Note that, by (5.137) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{s^{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{5.144}$$

Also, by the definition of E_j, we have

$$E_j \mid s^{j+1} = \langle\rangle, \quad \text{for each } j. \tag{5.145}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5. (5.146)

The induction base ($j = 0$) follows easily from (5.143), since $E_0 = \langle\rangle$.

Assume that (5.146) holds for a particular value of j. Since $s^{j+1} \in S$, by (5.115), we have

$$s^{j+1} \in Y, \tag{5.147}$$

and $s^{j+1} \in \mathrm{Act}(H)$. By applying (5.63) with 'p' $\leftarrow s^{j+1}$, and using (5.147), we also have $\mathrm{Act}(H \circ L_{s^{j+1}}) = \mathrm{Act}(H)$, and hence

$$s^{j+1} \in \mathrm{Act}(H \circ L_{s^{j+1}}). \tag{5.148}$$

Also, by (5.142),

- no events in E_j access any of s^{j+1}'s local variables. (5.149)

We use Lemma 5.2 twice in sequence in order to prove Claim 4. First, by P3, and applying (5.122), (5.132), and (5.136) with 's' $\leftarrow s^{j+1}$, it follows that there exists an event $e''_{s^{j+1}}$, such that

- $H' \circ L \circ \langle e''_{s^{j+1}}\rangle \in C$, and (5.150)

- $e''_{s^{j+1}} \sim e_{s^{j+1}}$. (5.151)

We now apply Lemma 5.2, with 'H' $\leftarrow H \circ L_{s^{j+1}}$, '$H'$' $\leftarrow H' \circ L$, 'G' $\leftarrow \langle\rangle$, '$RFS$' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e_{s^{j+1}}$, and 'e'_p' $\leftarrow e''_{s^{j+1}}$. Among the assumptions stated in Lemma 5.2, (5.5) and (5.7)–(5.9) follow from (5.150), (5.133), (5.151), and (5.148), respectively; (5.11) and (5.12) hold vacuously by 'G' $\leftarrow \langle\rangle$; (5.4), (5.6), and (5.10) follow by applying (5.122), (5.125), and (5.135), respectively, with 's' $\leftarrow s^{j+1}$; (5.13) follows by applying (5.62) with 'p' $\leftarrow s^{j+1}$, and using (5.147). It follows that

- $e''_{s^{j+1}}$ is critical in $H' \circ L \circ \langle e''_{s^{j+1}}\rangle$. (5.152)

Before applying Lemma 5.2 again, we establish the following preliminary assertions. Since $\mathrm{Fin}(H) \subseteq RFS$, by applying (5.125) with 's' $\leftarrow s^{j+1}$, it follows that

- RFS is a valid RF-set of $H \circ L_{s^{j+1}}$. (5.153)

We now establish a simple claim.

> **Claim 4-1:** If $e_{s^{j+1}}$ is a comparison event on a remote variable v, and if E_j contains a write to v, then $E_j \mid RFS$ also contains a write to v.
>
> **Proof of Claim:** If $e_{s^{j+1}}$ is a comparison event on v, then by (5.138) and Claim 3, we have $e'_{s^{j+1}} \in C_1(v) \cup C_2(v)$. By (5.142), no event in E_j may locally access v. Hence, by Claim 3, if an event e'_{s^k} (for some $k \leq j$) in E_j writes to v, then we have either $e'_{s^k} \in W(v)$ or $e'_{s^k} = SC(v)$. If $e'_{s^k} = SC(v)$, then since $s^k \in RFS$ holds by (5.141), Claim 4-1 is satisfied. On the other hand, if $e'_{s^k} \in W(v)$, then $W(v)$ is nonempty. Moreover, since all events in $W(v)$ are indexed before any events in $C_1(v) \cup C_2(v)$, E_j contains all events in $W(v)$. Thus, by (5.141) and the definition of LW, both E_j and $E_j \mid RFS$ contain $LW(v)$, an event that writes to v. □

We now apply Lemma 5.2 again, with 'H' $\leftarrow H' \circ L$, 'H'' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'e_p' $\leftarrow e''_{s^{j+1}}$, and 'e'_p' $\leftarrow e'_{s^{j+1}}$. Among the assumptions stated in Lemma 5.2, (5.4)–(5.7) and (5.11)–(5.13) follow from (5.150), (5.144), (5.143), (5.143), (5.145), (5.149), and (5.152), respectively; (5.10) is trivial; (5.8) follows from (5.151) and by applying (5.138) with 's^j' $\leftarrow s^{j+1}$; (5.9) follows from (5.140) and $s^{j+1} \in S$. Moreover, Assumption (A) follows from (5.146), and Assumption (B) follows from Claim 4-1.

It follows that $e'_{s^{j+1}}$ is critical in $H' \circ L \circ E_j \circ \langle e'_{s^{j+1}} \rangle = H' \circ L \circ E_{j+1}$, and that $H' \circ L \circ E_{j+1}$ satisfies RF5. □

We now show that RFS is a valid RF-set of $\overline{G}$. Condition RF5 was already proved in Claim 4.

- **RF1 and RF2:** Define E_j as in Claim 4. We establish RF1 and RF2 by induction on j, applying Lemma 5.3 at each step. At each step, we assume

  - $H' \circ L \circ E_j$ satisfies RF1 and RF2. (5.154)

  The induction base ($j = 0$) follows easily from (5.143), since $E_0 = \langle\rangle$.

  Assume that (5.154) holds for a particular value of j. Assume that $e'_{s^{j+1}}$ remotely accesses variable v.

  By Claim 3, if $e'_{s^{j+1}}$ remotely reads a variable v, then the following holds: $e'_{s^{j+1}} \in C_1(v) \cup C_2(v) \cup R(v)$; every event in $W(v)$ is contained in E_j; $\mathit{writer}(v, H' \circ L \circ E_j)$ is one of $LW(v)$ or $SC(v)$ or $\perp$. Therefore, by (5.141), we have the following:

  - if $e'_{s^{j+1}}$ remotely reads v, and if we let $q = \mathit{writer}(v, H' \circ L \circ E_j)$, then either $q = \perp$ or $q \in RFS$ holds. (5.155)

  We now apply Lemma 5.3, with 'H' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'e_p' $\leftarrow e'_{s^{j+1}}$, and 'v_{rem}' $\leftarrow v$. Among the assumptions stated in Lemma 5.3, (5.14), (5.15), (5.17), (5.19), and (5.21) follow from (5.144), (5.143), (5.154), (5.145), and (5.155), respectively; (5.16) follows from (5.140) and $s^{j+1} \in S$; (5.18) follows from (5.140) and (5.139); (5.20) follows from (5.140) and (5.142). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2.

- **RF3:** Consider a variable $v \in V$ and two different events f_p and g_q in $\overline{G}$. Assume that both p and q are in $\mathrm{Act}(\overline{G})$, $p \neq q$, that there exists a variable v such that $v \in \mathit{var}(f_p) \cup \mathit{var}(g_q)$, and that there exists a write to v in $\overline{G}$. Define $r = \mathit{writer}(v, \overline{G})$. Our proof obligation is to show that $r \in RFS$.

By (5.140), we have $\{p, q\} \subseteq S$. If there exists an event e'_s in E that remotely accesses v, then by Claim 3, $writer\_event(v, \overline{G})$ is either $SC(v)$ (if $SC(v) \neq \bot$) or $LW(v)$ (otherwise). (Since we assumed that there exists a write to v, they both cannot be $\bot$.) Thus, by (5.141), we have the following:

- if there exists an event e'_s in E such that e'_s remotely accesses v, then $r \in RFS$. (5.156)

We now consider three cases.

  – Consider the case in which both f_p and g_q are in $H' \circ L$.

    If there exists an event e'_s in E such that $v \in Wvar(e'_s)$, then we claim that v is remote to s. Assume, to the contrary, that v is local to s. Since at least one of p or q is different from s, without loss of generality, assume $p \neq s$. Since $p \in S$ and, by (5.115), $S \subseteq \mathrm{Act}(H)$, we have $p \notin \mathrm{Fin}(H)$. Thus, by (5.133) and by applying RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p in $H' \circ L$, we have $s \notin \mathrm{Act}(H' \circ L)$. However, by (5.140), $\mathrm{Act}(H' \circ L) = S$, which contradicts $s \in S$ (which follows from (5.139), since e'_s is an event of E). It follows that v is remote to s, and hence $r \in RFS$ by (5.156).

    On the other hand, if there exists no event e'_s in E such that $v \in Wvar(e'_s)$ holds, then we have $r = writer(v, H' \circ L)$. By (5.133) and applying RF3 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p and g_q in $H' \circ L$, we have $writer(v, H' \circ L) \in \mathrm{Fin}(H) \subseteq RFS$.

  – Consider the case in which f_p is in $H' \circ L$ and $g_q = e'_{s^k}$, for some $s^k \in S$. By (5.140) and our assumption that p and q are both in $\mathrm{Act}(\overline{G})$, we have $p \in \mathrm{Act}(H' \circ L)$ and $q \in \mathrm{Act}(H' \circ L)$. If v is local to q, then by (5.133), and by applying RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p in $H' \circ L$, we have $q \notin \mathrm{Act}(H' \circ L)$, a contradiction. Thus, v is remote to q, and hence $r \in RFS$ by (5.156).

  – Consider the case in which $f_p = e'_{s^j}$ and $g_q = e'_{s^k}$, for some s^j and s^k in S. Since v is remote to at least one of s^j or s^k, we have $r \in RFS$ by (5.156).

- **RF4:** By (5.60), (5.133), and (5.140), it easily follows that $\overline{G}$ satisfies RF4 with respect to RFS.

Therefore, we have established that

- RFS is a valid RF-set of $\overline{G}$. (5.157)

By (5.140) and (5.141), it follows that $\mathrm{Pmt}_{RFS}(\overline{G})$ consists of processes $owner(LW(v))$ and $owner(SC(v))$, for each variable $v \in V_{\mathrm{HC}}$. Thus, clearly $|\mathrm{Pmt}_{RFS}(\overline{G})| \leq 2|V_{\mathrm{HC}}|$ holds. Hence, from (5.67), we have

$$|\mathrm{Pmt}_{RFS}(\overline{G})| \leq n/(3 \log n). \tag{5.158}$$

We now let the processes in $\mathrm{Pmt}(\overline{G})$ finish their execution by inductively appending critical events of processes in $\mathrm{Pmt}(\overline{G})$, thus generating a sequence of computations G_0, G_1, ..., G_k (where $G_0 = \overline{G}$), satisfying the following:

- $G_j \in C$; (5.159)
- RFS is a valid RF-set of G_j; (5.160)
- $\mathrm{Pmt}(G_j) \subseteq \mathrm{Pmt}(\overline{G})$; (5.161)
- each process in $\mathrm{Inv}(G_j)$ executes exactly $c+1$ critical events in G_j; (5.162)
- the processes in $\mathrm{Pmt}(\overline{G})$ collectively execute exactly $|\mathrm{Pmt}(\overline{G})| \cdot (c+1) + j$ critical events in G_j; (5.163)
- $\mathrm{Inv}(G_{j+1}) \subseteq \mathrm{Inv}(G_j)$ and $|\mathrm{Inv}(G_{j+1)}| \geq |\mathrm{Inv}(G_j)| - 1$ if $j < k$; (5.164)
- $\mathrm{Fin}(G_j) \subsetneq RFS$ if $j < k$, and $\mathrm{Fin}(G_j) = RFS$ if $j = k$. (5.165)

At each induction step, we apply Lemma 5.6 to G_j in order to construct G_{j+1}, until $\mathrm{Fin}(G_j) = RFS$ is established, at which point the induction is completed. The induction is explained in detail below.

Induction base ($j = 0$): Since $G_0 = \overline{G}$, (5.159) and (5.160) follow from (5.137) and (5.157), respectively. Condition (5.161) is trivial.

By (5.51), (5.119), and (5.134), each process in S executes exactly c critical events in $H' \circ L$. Thus, by Claim 4, it follows that each process in S executes exactly $c+1$ critical events in $\overline{G}$. Since $\mathrm{Inv}(\overline{G}) \subseteq S$, $\overline{G}$ satisfies (5.162). Since $\mathrm{Pmt}(\overline{G}) \subseteq S$, $\overline{G}$ satisfies (5.163).

Induction step: At each step, we assume (5.159)–(5.163). If $\mathrm{Fin}(G_j) = RFS$, then (5.165) is satisfied and we finish the induction, by letting $k = j$.

Assume otherwise. We apply Lemma 5.6 with 'H' $\leftarrow G_j$. Assumptions (5.35)–(5.37) stated in Lemma 5.6 follow from (5.159), (5.160), and $\mathrm{Fin}(G_j) \neq RFS$. The lemma implies that a computation G_{j+1} exists satisfying (5.159)–(5.165), as shown below.

Condition (5.159) and (5.160) follow from (5.38) and (5.39), respectively. Since G_j satisfies (5.161), by (5.46), G_{j+1} also satisfies (5.161). Since $\mathrm{Inv}(G_{j+1)} \subseteq \mathrm{Inv}(G_j)$ by (5.40) and (5.41), by (5.43) and (5.47), and applying (5.162) to G_j, it follows that G_{j+1} satisfies (5.162). By (5.43)–(5.47), and applying (5.161) and (5.163) to G_j, it follows that G_{j+1} satisfies (5.163). Condition (5.164) follows from (5.40) and (5.41). Thus, the induction is established. □

We now show that $k < n/3$. Assume otherwise. By applying (5.163) to G_k, it follows that there exists a process $p \in \mathrm{Pmt}(\overline{G})$ such that p executes at least $c+1+k/|\mathrm{Pmt}(\overline{G})|$ critical events in G_k. Because $k \geq n/3$, by (5.158), p executes at least $c+1+\log n$ critical events in G_k. From (5.165) and $p \in \mathrm{Pmt}(\overline{G}) \subseteq \mathit{RFS}$, we get $p \in \mathrm{Fin}(G_k)$. Hence, by (5.160), and by applying RF4 to p in G_k, it follows that the last event by p is Exit_p. Therefore, G_k can be written as $F \circ \langle \mathit{Exit}_p \rangle \circ \cdots$, where F is a prefix of G_k such that p executes at least $c + \log n$ critical events in F. However, p and F then satisfy Proposition Pr1, a contradiction.

Finally, we show that G_k satisfies Proposition Pr2. The following derivation establishes (5.53).

$$\begin{aligned}
|\mathrm{Act}(G_k)| &= |\mathrm{Inv}_{\mathit{RFS}}(G_k)| && \{\text{by (5.165), } \mathit{RFS} = \mathrm{Fin}(G_k)\text{, thus } \mathrm{Act}(G_k) = \mathrm{Inv}_{\mathit{RFS}}(G_k)\} \\
&\geq |\mathrm{Inv}_{\mathit{RFS}}(G_0)| - k && \{\text{by repeatedly applying (5.164)}\} \\
&= |\mathrm{Act}(\overline{G}) - \mathit{RFS}| - k && \{\text{by the definition of ``Inv''; note that } \overline{G} = G_0\} \\
&= |S - \mathit{RFS}| - k && \{\text{by (5.140)}\} \\
&= |S - (\mathrm{Pmt}(\overline{G}) \cup \mathrm{Fin}(H))| - k && \{\text{because } \mathit{RFS} = \mathrm{Pmt}(\overline{G}) \cup \mathrm{Fin}(\overline{G})\text{, and} \\
& && \quad \mathrm{Fin}(\overline{G}) = \mathrm{Fin}(H) \text{ by (5.140)}\} \\
&= |S - \mathrm{Pmt}(\overline{G})| - k && \{\text{because } S \cap \mathrm{Fin}(H) = \{\} \text{ by (5.140)}\} \\
&= |(P_{\mathrm{HC}} - K) - \mathrm{Pmt}(\overline{G})| - k && \{\text{by (5.112)}\} \\
&\geq |P_{\mathrm{HC}}| - |K| - |\mathrm{Pmt}(\overline{G})| - k \\
&\geq \frac{|Y|}{2} - \frac{n}{6\log n} - \frac{n}{3\log n} - \frac{n}{3} && \{\text{by Case 2, (5.114), (5.158), and } k < n/3\} \\
&\geq \frac{n-1}{2} - \frac{n}{2\log n} - \frac{n}{3} && \{\text{by (5.56)}\} \\
&= \frac{n}{6} - \frac{n}{2\log n} - \frac{1}{2}.
\end{aligned}$$

```
$H_1 := \langle Enter_1,\ Enter_2,\ \ldots,\ Enter_N \rangle$;      $n_1 := N$;      $j := 1$;
repeat forever
LOOP INVARIANT: $H_j \in C$, $H_j$ is regular, $n_j = |\mathrm{Act}(H_j)|$, and each process in
                $\mathrm{Act}(H_j)$ executes exactly $j$ critical events in $H_j$.
    if $j > \log n_j - 1$ then
        let $k := j$, and exit the algorithm
    else      /* $j \le \log n - 1$ */
        apply Lemma 5.7 with 'H' $\leftarrow H_j$;
        if (Pr1) holds then
            let $k := j$, and exit the algorithm
        else      /* (Pr2) holds */
            — There exists a regular computation $G$ in $C$ such that $|\mathrm{Act}(G)| =$
            $\Omega(n_j/\log^2 n_j)$ and each process in $\mathrm{Act}(G)$ executes exactly $j+1$ critical
            events in $G$. Define $Z = \mathrm{Act}(G)$.
            $H_{j+1} := G$;  $n_{j+1} := |Z|$;  $j := j+1$
    fi fi
od
```

Figure 5.9: Algorithm for constructing $H_1, H_2, \ldots, H_k$.

Moreover, by (5.160) and (5.165), we have $\mathrm{Act}(G_k) = \mathrm{Inv}(G_k)$. Thus, by (5.115), (5.140), and (5.164), we have $\mathrm{Act}(G_k) \subseteq \mathrm{Inv}(\overline{G}) \subseteq \mathrm{Act}(\overline{G}) = S \subseteq \mathrm{Act}(H)$, which implies (5.52). Finally, (5.162) implies (5.54). Therefore, G_k satisfies Proposition Pr2. □

Theorem 5.2 *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exist a process p in P and a computation H in C such that $H \circ \langle Exit_p \rangle \in C$, H does not contain $Exit_p$, and p executes $\Omega(\log N/\log\log N)$ critical events in H, where $N = |P|$.*

Proof: Let $H_1 = \langle Enter_1,\ Enter_2,\ \ldots,\ Enter_N \rangle$, where $P = \{1, 2, \ldots, N\}$. By the definition of a mutual exclusion system, $H_1 \in C$. It is obvious that H_1 is regular and each process in $\mathrm{Act}(H) = P$ has exactly one critical event in H_1. Starting with H_1, we repeatedly apply Lemma 5.7 and construct a sequence of computations $H_1,\ H_2,\ \ldots,\ H_k$, such that each process in $\mathrm{Act}(H_j)$ has j critical events in H_j. The construction algorithm is shown in Figure 5.9.

For each computation H_j such that $1 \le j < k$, we have the following inequality:

$$n_{j+1} \ge \frac{cn_j}{\log^2 n_j} \ge \frac{cn_j}{\log^2 N},$$

where c is some fixed constant. This in turn implies

$$\log n_{j+1} \ge \log n_j - 2\log\log N + \log c. \tag{5.166}$$

By iterating over $1 \leq j < k$, and using $n_1 = N$, (5.166) implies

$$\log n_k \geq \log N - 2(k-1)\log\log N + (k-1)\log c. \tag{5.167}$$

We now consider two possibilities, depending on how the algorithm in Figure 5.9 terminates. First, suppose that H_k satisfies $k > \log n_k - 1$. Combining this inequality with (5.167), we have

$$k > \frac{\log N + 2\log\log N - \log c - 1}{2\log\log N - \log c + 1} = \Theta\left(\frac{\log N}{\log\log N}\right).$$

Therefore, each process in $\mathrm{Act}(H_k)$ executes $\Omega(\log N/\log\log N)$ critical events in H_k. By the Progress property, we can extend H_k to construct a computation that satisfies the theorem.

The other possibility is that $k \leq \log n_k - 1$ holds and H_k satisfies Proposition Pr1. In this case, a process p and a computation F exist such that $F \circ \langle \mathit{Exit}_p \rangle \in C$, F does not contain $\langle \mathit{Exit}_p \rangle$, and p executes at least $k + \log n_k$ critical events in F. By combining $k \leq \log n_k - 1$ with (5.167), we have

$$\log n_k \geq \frac{\log N + 4\log\log N - 2\log c}{2\log\log N - \log c + 1} = \Theta\left(\frac{\log N}{\log\log N}\right).$$

Therefore, computation F satisfies the theorem. $\square$

5.4 Constant-time Algorithm for LFCU Systems

In this section, we present a simple starvation-free mutual exclusion algorithm with $O(1)$ time complexity in LFCU systems, provided that each event that writes shared variables generates $O(1)$ interconnect traffic to update cached copies. In bus-based systems, this is a reasonable assumption, since an update message can be broadcast with a constant cost. On the other hand, in non-bus-based systems, an update may generate $\omega(1)$ interconnect traffic. However, the complexity involved in broadcasting a word-sized message in a non-bus-based network, while managing cache coherence, renders this approach exceedingly problematic. Indeed, we do not know of any commercial multiprocessor system that has a non-bus-based architecture and that uses a write-update cache protocol.

```
shared variables
    Check: $0..N-1$;
    Lock: boolean initially false;
    Promoted: array[$0..N-1$] of boolean initially false;
    Trying: array[$0..N-1$] of boolean initially false

process p ::    /* $0 \le p < N$ */
while true do
0:  Noncritical Section;
1:  Trying[p] := true;
2:  repeat /* null */ until (test-and-set(Lock) = false) $\vee$ Promoted[p];
3:  Promoted[p] := false;
4:  Trying[p] := false;
5:  Critical Section;
6:  s := Check;
7:  Check := s + 1 mod N;
8:  if Trying[s] then
9:      Promoted[s] := true
    else
10:     Lock := false
od
```

Figure 5.10: A mutual exclusion algorithm with $O(1)$ time complexity in LFCU systems. Each shared variable is remote to all processes.

The algorithm, which is shown in Figure 5.10, is a slight modification of the basic *test-and-set* lock. Recall that the atomic *test-and-set* primitive used in the algorithm is defined by the following pseudo-code (see Section 2.1).

```
test-and-set(bit: boolean) returns boolean
    if bit = false then bit := true; return false
    else return true
fi
```

In the algorithm, two shared variables are used, *Check* and *Lock*, and two shared arrays, *Promoted* and *Trying*. We assume that each of these variables is remote to all processes. Thus, our algorithm does not require the existence of locally-allocated shared memory as in a DSM system. Variable *Trying*[p] is *true* if and only if process p is executing within statements 2–4 of its entry section. Variable *Lock* is *false* if the *test-and-set* lock is available, and *true* otherwise. Variable *Promoted*[p] is *true* if and only if p has been given priority to enter its critical section, as explained below. Variable *Check* cycles through $0..N-1$ in order to determine the process to be given priority.

When a process p leaves its noncritical section, it sets *Trying*[p] = *true* at statement 1. It then enters the busy-waiting loop at statement 2. Process p may enter its critical section either by performing a successful *test-and-set* or by finding

Promoted$[p]$ = *true* at statement 2. (Note that we have defined *test-and-set* to return *false* when it succeeds.) The *Promoted* variables are used to prevent starvation: if other processes execute concurrently with p, then there is a possibility that p's *test-and-set* always fails. In order to prevent starvation, variable *Check* cycles through $0..N-1$, as seen in statements 6 and 7. (Note that these statements are executed before any other process is allowed to enter its critical section, so *Check* is incremented sequentially.) If p continues to wait at statement 2 while other processes execute their critical sections, then eventually (specifically, after at most N critical-section executions) some process reads *Check* $= p$ at statement 6, and establishes *Promoted*$[p]$ = *true* at statement 9. To prevent violations of the Exclusion property, *Lock* is not changed to *false* in this case. (This mechanism that gives priority to processes that might otherwise wait forever is rather similar to helping mechanisms used in wait-free algorithms [41]. Similar mechanism is also used in Algorithm T (presented in Section 8.3), some adaptive mutual exclusion algorithms [24, 30], and our recent work on timing-based mutual exclusion algorithms [48].)

It is straightforward to formalize these arguments and prove that the algorithm of Figure 5.10 is a correct, starvation-free mutual exclusion algorithm. We now prove that its time complexity is $O(1)$ per critical-section execution in LFCU systems. Clearly, time complexity is dominated by the number of remote memory references generated by statement 2. In an LFCU system, the *test-and-set* invocations in statement 2 generate $O(1)$ remote memory references. This is because a failed *test-and-set* generates a cached copy of *Lock*, and any subsequent update of *Lock* by a process at statement 10 updates this cached copy. The reads of *Promoted*$[p]$ in statement 2 also generate $O(1)$ remote memory references. In particular, the first read of *Promoted*$[p]$ creates a cached copy. If *Promoted*$[p]$ = *true*, then the loop terminates. If *Promoted*$[p]$ = *false*, then subsequent reads of *Promoted*$[p]$ are handled in-cache, until *Promoted*$[p]$ is updated by another process. Other processes update *Promoted*$[p]$ only by establishing *Promoted*$[p]$ = *true*. Once this is established, p's busy-waiting loop terminates. We conclude that the algorithm generates $O(1)$ remote memory references per critical-section execution in LFCU systems, as claimed.

5.5 Concluding Remarks

We have established a lower bound of $\Omega(\log N/\log\log N)$ remote memory references for mutual exclusion algorithms based on reads, writes, or comparison primitives; for algorithms with comparison primitives, this bound only applies in non-LFCU systems. Our bound improves an earlier lower bound of $\Omega(\log\log N/\log\log\log N)$ established by Cypher. We conjecture that $\Omega(\log N)$ is a tight lower bound for the class of algorithms and systems to which our lower bound applies; this conjecture remains an open issue.

It should be noted that Cypher's result guarantees that there exists no algorithm with *amortized* $\Theta(\log\log N/\log\log\log N)$ time complexity, while ours does not. This is because his bound is obtained by counting the total number of remote memory references in a computation, and by then dividing this number by the number of processes participating in that computation (see Theorem 2.16). In contrast, our result merely proves that there exists a computation H and a process p such that p executes $\Omega(\log N/\log\log N)$ critical events in H. Therefore, our result leaves open the possibility that the *average* number of remote memory references per process is less than $\Theta(\log N/\log\log N)$. We leave this issue for future research.

It is possible to generalize our lower-bound proof for systems with multi-valued and/or multi-variable comparison primitives. *Two-valued compare-and-swap* (2VCAS) and *double compare-and-swap* (DCAS) are examples of such primitives. 2VCAS uses two compare values *old*1 and *old*2 and two new values *new*1 and *new*2; a single variable v is compared to both and a new value is written to v if either comparison succeeds.

```
2VCAS($v$, $old1$, $old2$, $new1$, $new2$)
    $temp := v$;
    **if** $v = old1$ **then** $v := new1$
    **elseif** $v = old2$ **then** $v := new2$ **fi**;
    **return** $temp$
```

DCAS operates on two different variables u and v, using two associated compare values a and b, respectively; new values are assigned to u and v if and only if both $u = a$ and $v = b$ hold. In order to adapt our proof for systems in which such primitives are used, only the following change is needed: in the roll-forward strategy, for each variable, we select $O(1)$ processes with successful comparison events, instead of just one, and let all other processes execute unsuccessful comparison events. Therefore, we now roll $O(1)$ processes forward per variable. With this change, our asymptotic lower bound remains unchanged.[13] In Section 8.3, we show that there exists a class of (generalized) comparison primitives that includes 2VCAS for which a $\Theta(\log N/\log\log N)$ algorithm

[13]This argument does not apply to primitives that may compare a variable to an arbitrary number of values, or simultaneously compare an arbitrary number of variables. The existence of $O(1)$ algorithms for some *fetch-and-ϕ* primitives (see Sections 2.2.1 and 8.2) shows that at least some of these primitives *must* be excluded from our lower-bound proof.

is possible. (To the best of our knowledge, none of the primitives in this class has been implemented on a real machine. DCAS, which was supported on some generations of the Motorola 68000 processor family, is not in this class.) Thus, there exist comparison primitives for which our lower bound is tight.

In Chapter 6, we use proof techniques similar to those presented in this chapter to establish another lower bound that precludes the possibility of an $o(k)$ adaptive algorithm based on reads, writes, or comparison primitives, where k is either point or interval contention. The problem of designing an $O(\log k)$ algorithm using only reads and writes had been mentioned previously in at least two papers [15, 24]. The result of Chapter 6 shows that such an algorithm cannot exist.

CHAPTER 6

Time-complexity Lower Bound for Adaptive Mutual Exclusion*

In this chapter, we consider the RMR (remote-memory-reference) time complexity of adaptive mutual exclusion algorithms. In Chapter 4, we presented an $O(\min(k, \log N))$ adaptive algorithm, based only on reads and writes, where k is point contention. In Chapter 5, we showed that $\Omega(\log N / \log \log N)$ time is required for any mutual exclusion algorithm (adaptive or not) based on reads, writes, or comparison primitives. In this chapter, we establish a lower bound that precludes an $o(k)$ adaptive algorithm based on reads, writes, or comparison primitives, where k is either point or interval contention.

As defined earlier in Section 2.4, a mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes [15, 24, 30, 63, 78]. Two notions of contention have been considered in the literature: "interval contention" and "point contention" [2]. The *interval contention* over computation H is the number of processes that are active in H, *i.e.*, that execute outside of their noncritical sections. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. Throughout this chapter, k denotes the point/interval contention experienced by an arbitrary process while it is active. (In every computation considered in this chapter, point contention equals interval contention. Hence, the lower bound presented in this chapter applies to both point and interval contention.)

*The results presented in this chapter have been published in the following paper.
[46] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15. Lecture Notes in Computer Science 2180, Springer-Verlag, October 2001.

The $\Omega(\log N/\log\log N)$ lower bound presented in Chapter 5 does not mention k, so it tells us very little about time complexity under low contention. The best we can say is that $\Omega(\log k/\log\log k)$ RMRs are required. In particular, the $\Omega(\log N/\log\log N)$ lower bound is established by inductively considering longer and longer computations, the first of which involves N processes, and the last of which may involve fewer processes. If we start instead with k process, then a computation is obtained with $O(k)$ processes (and hence $O(k)$ point contention at each state) in which some process performs $\Omega(\log k/\log\log k)$ RMRs.

If $\Omega(\log N)$ is a tight lower bound, as conjectured in Chapter 5, then presumably a lower bound of $\Omega(\log k)$ would follow as well. This suggests two interesting possibilities: in all likelihood, either $\Omega(\min(k, \log N))$ is in fact a tight lower bound (*i.e.*, Algorithm A-LS, presented in Chapter 4, is optimal), or it is possible to design an adaptive algorithm with $O(\log k)$ time complexity (*i.e.*, $\Omega(\log k)$ is tight). Indeed, the problem of designing an $O(\log k)$ algorithm using only reads and writes has been mentioned in at least two papers [15, 24].

In this chapter, we show that an $O(\log k)$ algorithm does not exist. In particular, we prove the following.

> *Given any k, define $\bar{N} = \bar{N}(k) = (2k+4)^{2(2^k-1)}$. For any $N \geq \bar{N}$, and for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives, a computation exists involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ remote memory references to enter and exit its critical section.*

Our proof of this result utilizes techniques used in Chapter 5. The rest of the chapter is organized as follows. The key ideas of our lower-bound proof are sketched in Section 6.1. (We use the same system model developed in Chapter 5.) In Section 6.2, the proof is presented in detail. We conclude in Section 6.3.

6.1 Proof Strategy

In Section 6.2, we show that for any positive k, there exists some $\bar{N}$ such that, for any mutual exclusion system $\mathcal{S} = (C, P, V)$ with $|P| \geq \bar{N}$, there exists a computation H such that some process p experiences point contention k and executes at least k critical events to enter and exit its critical section. In this section, we sketch the key ideas of the proof.

As in Chapter 5, the proof focuses on a special class of computations, namely, regular computations. (For the definition of a regular computation, see Section 5.2.1.) Recall that a regular computation consists of events of two groups of processes, "active processes" and "finished processes." Informally, an active process is a process in its entry section, competing with other active processes; a finished process is a process that has executed its critical section once, and is in its noncritical section.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, and H be a computation in C. We define Act(H), the set of *active processes* in H, and Fin(H), the set of *finished processes* in H, as follows.

$$\begin{aligned}\mathrm{Act}(H) &= \{p \in P\colon H \mid p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is } \textit{not} \text{ in } H\} \\ \mathrm{Fin}(H) &= \{p \in P\colon H \mid p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is in } H\}\end{aligned} \qquad \square$$

As before, the proof proceeds by inductively constructing longer and longer regular computations, until the desired lower bound is attained. The regularity condition ensures that *no participating process has knowledge of any other process that is active*. This has two consequences. First, we can "erase" any active process (*i.e.*, remove its events from the computation) and still get a valid computation. Second, "most" active processes have a "next" critical event.

We begin with a brief discussion of similarities and differences between the proof of this chapter and that given in Chapter 5. As in Chapter 5, at each induction step, we start with a regular computation with n active processes, for some value of n. We then apply either the "erasing" or the "roll-forward" strategy, and construct a longer regular computation. We guarantee in all cases that $\Omega(\sqrt{n}/k)$ active processes remain, each of which executes one more critical event. Moreover, unlike Chapter 5, we also ensure that the new computation has at most two additional finished processes. Since an active process have knowledge of only finished processes, its perceived contention is bounded by the number of finished processes, and increases by at most two at each induction step. The induction continues until the desired lower bound of k critical events is achieved, at which point each active process perceives contention of $O(k)$. We now give a detailed proof overview.

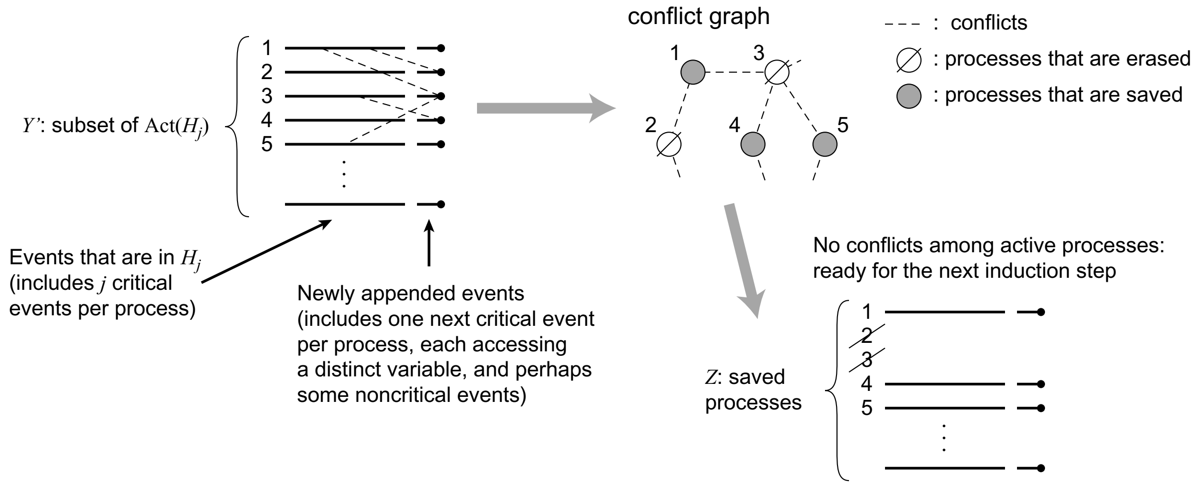
Proof overview. Our proof strategy is very similar to that of Chapter 5. Initially, we start with a regular computation H_1, where Act$(H_1) = P$, Fin$(H_1) = \{\}$, and each process has one critical event. We then inductively show that other longer computations exist, the last of which establishes our lower bound. Each computation is obtained by rolling forward or erasing some processes. We assume that P is large enough to ensure that enough non-erased processes remain after each induction step for the next step to be applied. The precise bound on $|P|$ is given in Theorem 6.1.

At the j^{th} induction step, we consider a computation H_j such that $\text{Act}(H_j)$ consists of n processes that execute j critical events each. We construct a regular computation H_{j+1} such that $\text{Act}(H_{j+1})$ consists of $\Omega(\sqrt{n}/k)$ processes, each of which executes $j+1$ critical events in H_{j+1}. The construction method, formally described in Lemma 6.1, is explained below. In constructing H_{j+1} from H_j, we may erase some processes and roll *at most two* processes forward. (This is the main difference between the proof of this chapter and that given in Chapter 5.) At the end of step $k-1$, we have a regular computation H_k in which each active process executes k critical events and $\text{Fin}(H_k) \leq 2(k-1)$. Since active processes have no knowledge of each other, we may erase all but one active process from H_k and obtain a valid computation. This computation has exactly one active process and at most $2(k-1)$ finished processes. Thus, its contention is at most $2k-1$. Moreover, the remaining active process performs k critical events, proving the desired lower bound.

We now describe how H_{j+1} is constructed from H_j. Let $n = |\text{Act}(H_j)|$. As shown in Lemma 5.5, among the n processes in $\text{Act}(H_j)$, at least $n-1$ processes can execute an additional critical event before entering its critical section. We call these events "next" critical events, and denote the corresponding set of processes by Y. We consider two cases, based on the variables remotely accessed by these next critical events.

Erasing strategy. Assume that there exist $\Omega(\sqrt{n})$ distinct variables that are remotely accessed by some next critical events. For each such variable v, we select one process whose next critical event accesses v. Let Y' be the set of selected processes. This situation is depicted in Figure 6.1. (Note that this situation is nearly identical to that shown in Figure 5.5.) We now eliminate remaining possible conflicts among processes in Y' by constructing a "conflict graph" $\mathcal{G}$ as follows.

Each process p in Y' is considered a vertex in $\mathcal{G}$. By induction, process p has j critical events in $\text{Act}(H_j)$ and one next critical event. For each of the $j+1$ critical events of p, **(i)** if the event accesses the same variable as the next critical event of some other process q, introduce edge (p, q). In addition, **(ii)** if the next critical event of p remotely accesses a local variable of q, also introduce edge (p, q).



Y': subset of Act(H_j)
1
2
3
4
5
Events that are in H_j (includes j critical events per process)
Newly appended events (includes one next critical event per process, each accessing a distinct variable, and perhaps some noncritical events)
conflict graph
--- : conflicts
: processes that are erased
: processes that are saved
No conflicts among active processes: ready for the next induction step
Z: saved processes

Figure 6.1: Erasing strategy. For simplicity, processes in Fin(H_j) are not shown.

Since each process in Y' accesses a distinct remote variable in its next critical event, it is clear that each process generates at most $j + 1$ edges by rule (i) and at most one edge by rule (ii). By applying Turán's theorem (Theorem 5.1), we can find a subset Z of Y' such that $|Z| = \Omega(\sqrt{n}/j)$ and their critical events do not conflict with each other. By retaining Z and erasing all other active processes, we can eliminate all conflicts. Thus, we can construct H_{j+1}.

Roll-forward strategy. Assume that the number of distinct variables that are remotely accessed by some next critical events is $O(\sqrt{n})$. This situation is depicted in Figure 6.2. Since there are $\Theta(n)$ next critical events, there exists a variable v that is remotely accessed by next critical events of $\Omega(\sqrt{n})$ processes. Let Y_v be the set of these processes. First, we retain Y_v and erase all other active processes. Let the resulting computation be H'. We then arrange the next critical events of Y_v by placing write, comparison, and read events in that order. Then, all next write events (of v), except for the last one, are overwritten by subsequent writes, and hence cannot create any information flow. (That is, even if some other process later reads v, it cannot gather any information of these "next" writers, except for the last one.) Furthermore, we can arrange comparison events such that at most one of them succeeds, as follows.

Assume that the value of v is α after all the next write events are executed. We first append all comparison events with an operation that can be written as compare(v, β) such that $\beta \neq \alpha$. These comparison events must fail. We then append all the remaining

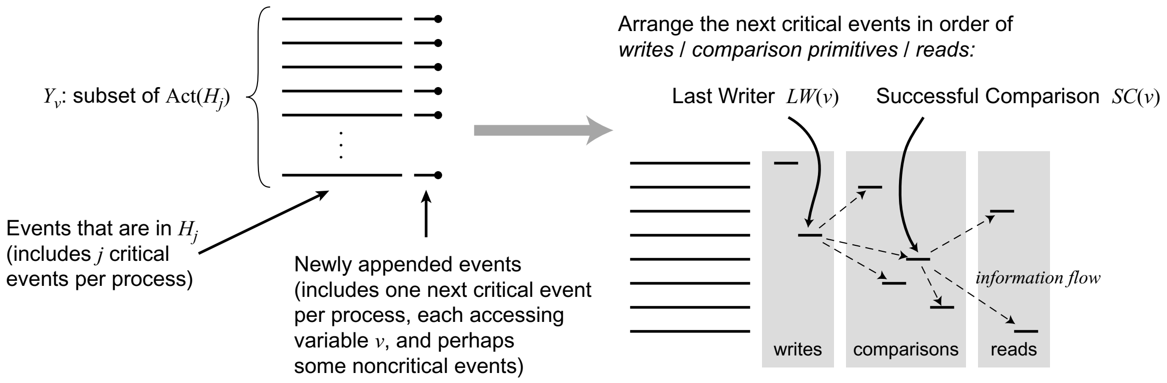

Y_v: subset of Act(H_j)
Events that are in H_j (includes j critical events per process)
Newly appended events (includes one next critical event per process, each accessing variable v, and perhaps some noncritical events)
Arrange the next critical events in order of *writes / comparison primitives / reads:*
Last Writer $LW(v)$
Successful Comparison $SC(v)$
information flow
writes
comparisons
reads

Figure 6.2: Roll-forward strategy. For simplicity, processes in $\mathrm{Fin}(H_j)$ are not shown.

comparison events, namely, events with operation compare(v, α). The first successful event among them (if any) changes the value of v. Thus, all subsequent comparison events must fail. (This situation is very similar to that shown in Figure 5.6, except that we need to consider only one variable v here.)

Thus, among the next events (that are not erased so far), the only information flow that arises is from the "last writer" event $LW(v)$ and from the "successful comparison" event $SC(v)$ to all other next comparison and read events of v.

Let p_{LW} and p_{SC} be the owner of $LW(v)$ and $SC(v)$, respectively. (Depending on the computation, we may have only one of them, or neither.) We then roll p_{LW} and p_{SC} forward by generating a regular computation G from H' such that $\mathrm{Fin}(G) = \mathrm{Fin}(H') \cup \{p_{\mathrm{LW}}, p_{\mathrm{SC}}\}$.

If either p_{LW} or p_{SC} executes at least k critical events before reaching its noncritical section, then the $\Omega(k)$ lower bound easily follows. Therefore, we can assume that either of p_{LW} and p_{SC} performs fewer than k critical events while being rolled forward. Each critical event of p_{LW} or p_{SC} that is appended to H' may generate information flow only if it reads a variable v that is written by another process in H'. Condition RF3 (given on page 102) guarantees that if there are multiple processes that write to v, the last writer in H' is not active. Because information flow from an inactive process is allowed, a conflict arises only if there is a single process that writes to v in H'. Thus, each critical event of p_{LW} or p_{SC} conflicts with at most one process in Y_v, and hence can erase at most one process. (Appending a noncritical event to H' cannot cause any processes to be erased. In particular, if a noncritical remote read by p_{LW} (respectively, p_{SC}) is appended, then p_{LW} (respectively, p_{SC}) must have previously read the same variable. By RF3, if the last writer is another process, then that process is not active.)

Therefore, the entire roll-forward procedure erases fewer than $2k$ processes from $\mathrm{Act}(H') = Y_v$. We can assume $|P|$ is sufficiently large to ensure that $\sqrt{n} > 4k$. This ensures that $\Omega(\sqrt{n})$ processes survive after the entire procedure. (Actually, as seen in Theorem 6.1, we only ensure that $\Omega(\sqrt{n}/k)$ processes survive, in order to simplify bookkeeping. This results in a larger bound on $|P|$. However, it is only of secondary interest, since our main goal is a lower bound on the number of critical events.) Thus, we can construct H_{j+1}.

6.2 Detailed Lower-bound Proof

In this section, we present our lower-bound theorem. Throughout this section, we assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$. The following lemma provides the induction step that leads to the lower bound in Theorem 6.1.

Lemma 6.1 Let k be a positive integer, and H be a computation. Assume the following:

- $H \in C$, and (6.1)
- H is regular (*i.e.*, $\mathrm{Fin}(H)$ is a valid RF-set of H). (6.2)

Define $n = |\mathrm{Act}(H)|$. Also assume that

- $n > 1$, and (6.3)
- each process in $\mathrm{Act}(H)$ executes exactly c critical events in H. (6.4)

Then, one of the following propositions is true.

Pr1: There exist a process p in $\mathrm{Act}(H)$ and a computation F in C such that

- $F \circ \langle \mathit{Exit}_p \rangle \in C$;
- F does not contain $\langle \mathit{Exit}_p \rangle$;
- at most $|\mathrm{Fin}(H) + 2|$ processes participate in F;
- p executes at least k critical events in F.

Pr2: There exists a regular computation G in C such that

- $\mathrm{Act}(G) \subseteq \mathrm{Act}(H)$; (6.5)
- $|\mathrm{Fin}(G)| \le |\mathrm{Fin}(H) + 2|$; (6.6)
- $|\mathrm{Act}(G)| \ge \min(\sqrt{n}/(2c+3),\ \sqrt{n} - 2k - 3)$; (6.7)
- each process in $\mathrm{Act}(G)$ executes exactly $(c+1)$ critical events in G. (6.8)

Proof: We first apply Lemma 5.5. Assumptions (5.28) and (5.29) stated in Lemma 5.5 follow from (6.1) and (6.2), respectively. It follows that there exists a set of processes Y such that

- $Y \subseteq \mathrm{Act}(H)$, and (6.9)
- $n - 1 \leq |Y| \leq n$, (6.10)

and for each process $p \in Y$, there exist a computation L_p and an event e_p by p, such that

- $H \circ L_p \circ \langle e_p \rangle \in C$; (6.11)
- L_p is a p-computation; (6.12)
- L_p contains no critical events in $H \circ L_p$; (6.13)
- $e_p \notin \{\mathit{Enter}_p, \mathit{CS}_p, \mathit{Exit}_p\}$; (6.14)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (6.15)
- e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$. (6.16)

For each $p \in Y$, by (6.12), (6.13), and $p \in Y \subseteq \mathrm{Act}(H)$, we have

$$\mathrm{Act}(H \circ L_p) = \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H \circ L_p) = \mathrm{Fin}(H). \tag{6.17}$$

By (6.3) and (6.10), Y is nonempty.

If Proposition Pr1 is satisfied by any process in Y, then the theorem is clearly true. Thus, we will assume, throughout the remainder of the proof, that there is no process in Y that satisfies Pr1. Define $\mathcal{E}_H$ as the set of critical events in H of processes in Y.

$$\mathcal{E}_H = \{f_q \text{ in } H\text{: } f_q \text{ is critical in } H \text{ and } q \in Y\}. \tag{6.18}$$

Define $\mathcal{E} = \mathcal{E}_H \cup \{e_p\text{: } p \in Y\}$, *i.e.*, the set of all "past" and "next" critical events of processes in Y. From (6.4), (6.9), and (6.10), it follows that

$$|\mathcal{E}| = (c + 1)|Y| \leq (c + 1)n. \tag{6.19}$$

Define V_{next} as the set of variables remotely accessed by some "next" critical events:

$$V_{\text{next}} = \{v \in V\text{: there exists } p \in Y \text{ such that } e_p \text{ remotely accesses } v\}. \tag{6.20}$$

We consider two cases, depending on the size of V_{next}.

Case 1: $|V_{\text{next}}| \geq \sqrt{n}$ (erasing strategy)
— In this case, we construct a subset Y' of Y by selecting one process for each variable in V_{next}. Clearly, $|Y'| = |V_{\text{next}}|$. We then construct a "conflict graph" $\mathcal{G}$, where each vertex is a process in Y'. By applying Theorem 5.1, we can find a subset Z of Y' such that their critical events do not conflict with each other. By applying Lemma 5.1 to H and $Z \cup \text{Fin}(H)$, and extending the resulting computation H' with next critical events, we construct a computation G that satisfies Proposition Pr2.

By definition, for each variable v in V_{next}, there exists a process p in Y such that e_p remotely accesses v. Therefore, we can arbitrarily select one such process for each variable v in V_{next} and construct a set Y' of processes such that

- $Y' \subseteq Y$, (6.21)
- if $p \in Y'$, $q \in Y'$ and $p \neq q$, then e_p and e_q access different remote variables, and (6.22)
- $|Y'| = |V_{\text{next}}| \geq \sqrt{n}$. (6.23)

We now construct an undirected graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y'. To each process y in Y' and each variable $v \in \mathit{var}(e_y)$ that is remote to y, we apply the following rules.

- **R1:** If v is local to a process z in Y', then introduce edge $\{y, z\}$.
- **R2:** If there exists an event $f_p \in \mathcal{E}$ that remotely accesses v, and if $p \in Y'$, then introduce edge $\{y, p\}$.

Because each variable is local to at most one process, and since (by the Atomicity property, given on page 90) an event can access at most one remote variable, Rule R1 can introduce at most one edge per process. Since, by (6.4), y executes exactly c critical events in H, by (6.22), Rule R2 can introduce at most c edges per process.

Combining Rules R1 and R2, at most $c+1$ edges are introduced per process. Since each edge is counted twice (for each of its endpoints), the average degree of $\mathcal{G}$ is at most $2(c+1)$. Hence, by Theorem 5.1, there exists an independent set Z such that

$$Z \subseteq Y', \quad \text{and} \tag{6.24}$$

$$|Z| \geq |Y'|/(2c+3) \geq \sqrt{n}/(2c+3), \tag{6.25}$$

where the latter inequality follows from (6.23).

(The rest of Case 1 is nearly identical to Case 1 in the proof of Lemma 5.7. We present the detailed argument here for the sake of completeness.)

Next, we construct a computation G, satisfying Proposition Pr2, such that $\mathrm{Act}(G) = |Z|$.

Define H' as

$$H' = H \mid (Z \cup \mathrm{Fin}(H)). \tag{6.26}$$

By (6.9), (6.21), and (6.24), we have

$$Z \subseteq Y' \subseteq Y \subseteq \mathrm{Act}(H), \tag{6.27}$$

and hence,

$$\mathrm{Act}(H') = Z \subseteq \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{6.28}$$

We now apply Lemma 5.1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$ and 'Y' $\leftarrow Z \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (6.1) and (6.2), respectively; (5.3) is trivial. It follows that

- $H' \in C$, (6.29)
- $\mathrm{Fin}(H)$ is a valid RF-set of H', and (6.30)
- an event in H' is critical if and only if it is also critical in H. (6.31)

Our goal now is to show that H' can be extended so that each process in Z has one more critical event. By (6.28), (6.30), and by the definition of a finished process,

$$\mathrm{Inv}_{\mathrm{Fin}(H)}(H') = \mathrm{Act}(H') = Z. \tag{6.32}$$

For each $z \in Z$, define F_z as

$$F_z = (H \circ L_z) \mid (Z \cup \mathrm{Fin}(H)). \tag{6.33}$$

By (6.27), we have $z \in Y$. Thus, applying (6.11), (6.12), (6.13), and (6.15) with 'p' $\leftarrow z$, it follows that

- $H \circ L_z \circ \langle e_z \rangle \in C$; (6.34)
- L_z is a z-computation; (6.35)
- L_z contains no critical events in $H \circ L_z$; (6.36)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_z$. (6.37)

By P1 (given on page 92), (6.34) implies

$$H \circ L_z \in C. \tag{6.38}$$

We now apply Lemma 5.1, with 'H' $\leftarrow H \circ L_z$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow Z \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (6.38) and (6.37), respectively; (5.3) is trivial. It follows that

- $F_z \in C$, and (6.39)
- an event in F_z is critical if and only if it is also critical in $H \circ L_z$. (6.40)

Since $z \in Z$, by (6.26), (6.33), and (6.35), we have

$$F_z = H' \circ L_z.$$

Hence, by (6.36) and (6.40),

- L_z contains no critical events in $F_z = H' \circ L_z$. (6.41)

Let $m = |Z|$ and index the processes in Z as $Z = \{z_1,\ z_2,\ \ldots,\ z_m\}$. Define $L = L_{z_1} \circ L_{z_2} \circ \cdots \circ L_{z_m}$. We now use Lemma 5.4, with 'H' $\leftarrow H'$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow Z$, and 'p_j' $\leftarrow z_j$ for each $j = 1, \ldots, m$. Among the assumptions stated in Lemma 5.4, (5.22)–(5.24) follow from (6.29), (6.30), and (6.32), respectively; (5.25)–(5.27) follow from (6.35), (6.39), and (6.41), respectively, with 'z' $\leftarrow z_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (6.42)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (6.43)
- L contains no critical events in $H' \circ L$. (6.44)

To this point, we have successfully appended a (possibly empty) sequence of non-critical events for each process in Z. It remains to append a "next" critical event for each such process. Note that, by (6.35) and the definition of L,

- L is a Z-computation. (6.45)

Thus, by (6.28) and (6.44), we have

$$\mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = Z \quad \wedge \quad \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{6.46}$$

By (6.26) and the definition of L, it follows that

- for each $z \in Z$, $(H \circ L_z) \mid (\{z\} \cup \mathrm{Fin}(H)) = (H' \circ L) \mid (\{z\} \cup \mathrm{Fin}(H))$. (6.47)

In particular, $H \circ L_z$ and $H' \circ L$ are equivalent with respect to z. Therefore, by (6.34), (6.42), and repeatedly applying P3, it follows that, for each $z_j \in Z$, there exists an event e'_{z_j}, such that

- $G \in C$, where $G = H' \circ L \circ E$ and $E = \langle e'_{z_1},\ e'_{z_2},\ \ldots,\ e'_{z_m} \rangle$; (6.48)
- $e'_{z_j} \sim e_{z_j}$. (6.49)

By the definition of E,

- E is a Z-computation. (6.50)

By (6.14), (6.46), and (6.49), we have

$$\mathrm{Act}(G) = \mathrm{Act}(H' \circ L) = Z \quad \wedge \quad \mathrm{Fin}(G) = \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H). \tag{6.51}$$

By (6.14), (6.16), and (6.49), it follows that for each $z_j \in Z$, both e_{z_j} and e'_{z_j} access a common remote variable, say, v_j. Since Z is an independent set of $\mathcal{G}$, by Rules R1 and R2, we have the following:

- for each $z_j \in Z$, v_j is not local to any process in Z; (6.52)
- $v_j \neq v_k$, if $j \neq k$.

Combining these two, we also have:

- for each $z_j \in Z$, no event in E other than e'_{z_j} accesses v_j (either locally or remotely). (6.53)

We now establish two claims.

Claim 1: For each $z_j \in Z$, if we let $q = \mathit{writer}(v_j, H' \circ L)$, then one of the following holds: $q = \perp$, $q = z_j$, or $q \in \mathrm{Fin}(H)$.

Proof of Claim: It suffices to consider the case when $q \neq \perp$ and $q \neq z_j$ hold, in which case there exists an event f_q by q in $H' \circ L$ that writes to v_j. By (6.26) and (6.45), we have $q \in Z \cup \mathrm{Fin}(H)$. We claim that $q \in \mathrm{Fin}(H)$ holds in this case. Assume, to the contrary,

$$q \in Z. \tag{6.54}$$

We consider two cases. First, if f_q is a critical event in $H' \circ L$, then by (6.44), f_q is an event of H', and hence, by (6.31), f_q is also a critical event in H. By (6.27) and (6.54), we have $q \in Y$. Thus, by (6.18), we have $f_q \in \mathcal{E}_H$, and hence $f_q \in \mathcal{E}$ holds by definition. By (6.52) and (6.54), v_j is remote to q. Thus, f_q *remotely* writes v_j. By (6.54) and $z_j \in Z$, we have

$$\{q, z_j\} \subseteq Z, \tag{6.55}$$

which implies $\{q, z_j\} \subseteq Y'$ by (6.24). From this, our assumption of $q \neq z_j$, and by applying Rule R2 with 'y' $\leftarrow z_j$ and 'f_p' $\leftarrow f_q$, it follows that edge $\{q, z_j\}$ exists in $\mathcal{G}$. However, (6.55) then implies that Z is not an independent set of $\mathcal{G}$, a contradiction.

Second, assume that f_q is a noncritical event in $H' \circ L$. Note that, by (6.52) and (6.54), v_j is remote to q. Hence, by the definition of a critical event, there exists a critical event $\bar{f}_q$ by q in $H' \circ L$ that remotely writes to v_j. However, this leads to contradiction as shown above. $\square$

Claim 2: Every event in E is critical in G. Also, G satisfies RF5 with 'RFS' $\leftarrow$ Fin(H).

Proof of Claim: Define $E_0 = \langle\rangle$; for each positive j, define E_j to be $\langle e'_{z_1}, e'_{z_2}, \ldots, e'_{z_j}\rangle$, a prefix of E. We prove the claim by induction on j, applying Lemma 5.2 at each step. Note that, by (6.48) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{z_{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{6.56}$$

Also, by the definition of E_j, we have

$$E_j \mid z_{j+1} = \langle\rangle, \quad \text{for each } j. \tag{6.57}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5 with 'RFS' $\leftarrow$ Fin(H). (6.58)

The induction base ($j = 0$) follows easily from (6.43), since $E_0 = \langle\rangle$.

Assume that (6.58) holds for a particular value of j. Since $z_{j+1} \in Z$, by (6.27), we have

$$z_{j+1} \in Y, \tag{6.59}$$

and $z_{j+1} \in \text{Act}(H)$. By applying (6.17) with 'p' $\leftarrow z_{j+1}$, and using (6.59), we also have $\text{Act}(H \circ L_{z_{j+1}}) = \text{Act}(H)$, and hence

$$z_{j+1} \in \text{Act}(H \circ L_{z_{j+1}}). \tag{6.60}$$

By (6.57), if any event e'_{z_k} in E_j accesses a local variable v of z_{j+1}, then e'_{z_k} accesses v *remotely*, and hence $v = v_k$ by definition. However, by (6.52), v_k cannot be local to z_{j+1}. It follows that

- no events in E_j access any of z_{j+1}'s local variables. (6.61)

We now apply Lemma 5.2, with 'H' $\leftarrow H \circ L_{z_{j+1}}$, '$H'$' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'RFS' $\leftarrow \text{Fin}(H)$, 'e_p' $\leftarrow e_{z_{j+1}}$, and 'e'_p' $\leftarrow e'_{z_{j+1}}$. Among the assumptions stated in Lemma 5.2, (5.5), (5.7), (5.9), (5.11), and (5.12) follow from (6.56), (6.43), (6.60), (6.57), and (6.61), respectively; (5.8) follows by applying (6.49) with 'z_j' $\leftarrow z_{j+1}$; (5.6) and (5.10) follow by applying (6.37) and (6.47), respectively, with 'z' $\leftarrow z_{j+1}$; and (5.4) and (5.13) follow by applying (6.11) and (6.16), respectively, with 'p' $\leftarrow z_{j+1}$, and using (6.59). Moreover, Assumption (A) follows from (6.58), and Assumption (B) is satisfied vacuously (with 'v' $\leftarrow v_{j+1}$) by (6.53).

It follows that $e'_{z_{j+1}}$ is critical in $H' \circ L \circ E_j \circ \langle e'_{z_{j+1}} \rangle = H' \circ L \circ E_{j+1}$, and that $H' \circ L \circ E_{j+1}$ satisfies RF5 with 'RFS' $\leftarrow \text{Fin}(H)$. $\square$

We now claim that $\text{Fin}(H)$ is a valid RF-set of G. Condition RF5 was already proved in Claim 2.

- **RF1 and RF2:** Define E_j as in Claim 2. We establish RF1 and RF2 by induction on j, applying Lemma 5.3 at each step. At each step, we assume

  - $H' \circ L \circ E_j$ satisfies RF1 and RF2 with 'RFS' $\leftarrow \text{Fin}(H)$. (6.62)

  The induction base ($j = 0$) follows easily from (6.43), since $E_0 = \langle\rangle$.

  Assume that (6.62) holds for a particular value of j. Note that, by (6.53), we have $\textit{writer}(v_{j+1}, H' \circ L \circ E_j) = \textit{writer}(v_{j+1}, H' \circ L)$. Thus, by (6.46) and Claim 1,

  - if we let $q = \mathit{writer}(v_{j+1}, H' \circ L \circ E_j)$, then one of the following holds: $q = \perp$, $q = z_{j+1}$, or $q \in \mathrm{Fin}(H) = \mathrm{Fin}(H' \circ L)$. (6.63)

  We now apply Lemma 5.3, with 'H' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e'_{z_{j+1}}$, and 'v_{rem}' $\leftarrow v_{j+1}$. Among the assumptions stated in Lemma 5.3, (5.14), (5.15), (5.17), (5.19), and (5.21) follow from (6.56), (6.43), (6.62), (6.57), and (6.63), respectively; (5.16) follows from (6.46) and $z_{j+1} \in Z$; (5.18) follows from (6.46) and (6.50); (5.20) follows from (6.52) and (6.46). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$.

- **RF3:** Consider a variable $v \in V$ and two different events f_q and g_r in G. Assume that both q and r are in $\mathrm{Act}(G)$, $q \neq r$, and that there exists a variable v such that $v \in \mathit{var}(f_q) \cap \mathit{var}(g_r)$. (Note that, by (6.51), $\{q, r\} \subseteq Z$.) We claim that these conditions can actually never arise simultaneously, which implies that G vacuously satisfies RF3.

  Since v is remote to at least one of q or r, without loss of generality, assume that v is remote to q. We claim that there exists an event $\bar{f}_q$ in $\mathcal{E}$ that accesses the same variable v. If f_q is an event of E, we have $f_q = e'_{z_j}$ for some $z_j \in Z$, and $e_{z_j} \in \mathcal{E}$ holds by definition; define $\bar{f}_q = e_{z_j}$ in this case. If f_q is a noncritical event in $H' \circ L$, then by definition of a critical event, there exists a critical event $\bar{f}_q$ in $H' \circ L$ that remotely accesses v. If f_q is a critical event in $H' \circ L$, then define $\bar{f}_q = f_q$. (Note that, if $\bar{f}_q$ is a critical event in $H' \circ L$, then by (6.31) and (6.44), $\bar{f}_q$ is also a critical event in H, and hence, by $q \in Z$, (6.27), and the definition of $\mathcal{E}$, we have $\bar{f}_q \in \mathcal{E}$.)

  It follows that, in each case, there exists an event $\bar{f}_q \in \mathcal{E}$ that remotely accesses v. If v is local to r, then by Rule R1, $\mathcal{G}$ contains the edge $\{q, r\}$. On the other hand, if v is remote to r, then we can choose an event $\bar{g}_r \in \mathcal{E}$ that remotely accesses v, in the same way as shown above. Hence, by Rule R2, $\mathcal{G}$ contains the edge $\{q, r\}$. Thus, in either case, p and q cannot simultaneously belong to Z, a contradiction.

- **RF4:** By (6.43) and (6.51), it easily follows that G satisfies RF4 with respect to $\mathrm{Fin}(H)$.

Finally, we claim that G satisfies Proposition Pr2. By (6.51), we have $\mathrm{Act}(G) = Z \subseteq \mathrm{Act}(H)$, so G satisfies (6.5) and (6.6). By (6.25), we have (6.7). By (6.4), (6.31), and (6.44), each process in Z executes exactly c critical events in $H' \circ L$. Thus, by Claim 2, G satisfies (6.8).

Case 2: $|V_{\text{next}}| \le \sqrt{n}$ (roll-forward strategy)

— *In this case, there exists a variable v that is remotely accessed by next critical events of at least $\sqrt{n}-1$ processes. Let Y_v be the set of these processes. We retain Y_v and erase all other active processes. Let the resulting computation be H'. We then roll forward processes p_{LW} and p_{SC} of Y_v to generate a regular computation G. If either p_{LW} or p_{SC} executes k or more critical events before finishing its execution, the resulting computation satisfies Proposition Pr1. Otherwise, fewer than $2k$ processes are erased during the procedure, which makes G satisfy Proposition Pr2, with at least $\sqrt{n}-2k$ active processes.*

For each variable v_j in V_{next}, define $Y_{v_j} = \{p \in Y\colon e_p \text{ remotely accesses } v_j\}$. By (6.10) and (6.20), $|V_{\text{next}}| \le \sqrt{n}$ implies that there exists a variable v in V_{next} such that $|Y_v| \ge (n-1)/\sqrt{n}$ holds. (In the rest of Case 2, we consider v a fixed variable.) Then, the following holds:

$$|Y_v| \ge (n-1)/\sqrt{n} > \sqrt{n}-1 \tag{6.64}$$

(The rest of Case 2 is nearly identical to Case 2 in the proof of Lemma 5.7, except that we consider a single variable v. We present the detailed argument here for the sake of completeness.)

Define

$$H' = H \mid (Y_v \cup \text{Fin}(H)). \tag{6.65}$$

Using $Y_v \subseteq Y \subseteq \text{Act}(H)$, we also have

$$\text{Act}(H') = Y_v \subseteq \text{Act}(H) \quad \wedge \quad \text{Fin}(H') = \text{Fin}(H). \tag{6.66}$$

We now apply Lemma 5.1, with 'RFS' $\leftarrow \text{Fin}(H)$ and 'Y' $\leftarrow Y_v \cup \text{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (6.1) and (6.2), respectively; (5.3) is trivial. It follows that

- $H' \in C$, (6.67)
- $\text{Fin}(H)$ is a valid RF-set of H', and (6.68)
- an event in H' is critical if and only if it is also critical in H. (6.69)

Our goal now is to show that H' can be extended to a computation $\overline{G}$ (defined later), so that each process in Y_v has one more critical event. By (6.66), (6.68), and by the definition of a finished process,

$$\text{Inv}_{\text{Fin}(H)}(H') = \text{Act}(H') = Y_v. \tag{6.70}$$

For each $s \in Y_v$, define F_s as

$$F_s = (H \circ L_s) \mid (Y_v \cup \mathrm{Fin}(H)). \tag{6.71}$$

Since $Y_v \subseteq Y$, we have $s \in Y$. Thus, applying (6.11), (6.12), (6.13), and (6.15) with 'p' $\leftarrow s$, it follows that

- $H \circ L_s \circ \langle e_s \rangle \in C$; (6.72)
- L_s is an s-computation; (6.73)
- L_s contains no critical events in $H \circ L_s$; (6.74)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_s$. (6.75)

By P1, (6.72) implies

$$H \circ L_s \in C. \tag{6.76}$$

We now apply Lemma 5.1, with 'H' $\leftarrow H \circ L_s$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow Y_v \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (5.1) and (5.2) follow from (6.76) and (6.75), respectively; (5.3) is trivial. It follows that

- $F_s \in C$, and (6.77)
- an event in F_s is critical if and only if it is also critical in $H \circ L_s$. (6.78)

Since $s \in Y_v$, by (6.65), (6.71), (6.73), and (6.77), we have

- $F_s = H' \circ L_s \in C$. (6.79)

Hence, by (6.74) and (6.78),

- L_s contains no critical events in $F_s = H' \circ L_s$. (6.80)

We now show that the events in $\{L_s \colon s \in Y_v\}$ can be "merged" by applying Lemma 5.4. We arbitrarily index Y_v as $\{s_1,\ s_2,\ \ldots,\ s_m\}$, where $m = |Y_v|$. (Later, we construct a specific indexing of Y_v to reduce information flow.) Let $L = L_{s_1} \circ L_{s_2} \circ \cdots \circ L_{s_m}$. Apply Lemma 5.4, with 'H' $\leftarrow H'$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow Y_v$, and 'p_j' $\leftarrow s_j$ for each $j = 1, \ldots, m$. Among the assumptions stated in Lemma 5.4, (5.22)–(5.24) follow from (6.67), (6.68), and (6.70), respectively; (5.25)–(5.27) follow from (6.73), (6.79), and (6.80), respectively, with 's' $\leftarrow s_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (6.81)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (6.82)
- L contains no critical events in $H' \circ L$. (6.83)

By (6.65) and the definition of L, we also have,

- for each $s \in Y_v$, $(H \circ L_s) \mid (\{s\} \cup \mathrm{Fin}(H)) = (H' \circ L) \mid (\{s\} \cup \mathrm{Fin}(H))$; (6.84)
- for each $s \in Y_v$, $(H' \circ L) \mid s = (H \circ L_s) \mid s$. (6.85)

We now re-index the processes in Y_v so that information flow among them is minimized. We place next critical events of Y_v by placing write, comparison, and read events in that order. Furthermore, we can arrange comparison events such that at most one of them succeeds, as explained in Section 6.1. (Formally, the re-indexing method is very similar to that given in Figure 5.8, and illustrated in Figure 5.6, except that we now consider only one variable v.) Let $(s^1,\ s^2,\ \ldots,\ s^m)$ be the indexing of Y_v thus constructed, and E be the appended computation that consists of next critical events by processes in Y. Then, we have the following:

- $\overline{G} \in C$, where $\overline{G} = H' \circ L \circ E$ and $E = \langle e'_{s^1},\ e'_{s^2},\ \ldots,\ e'_{s^m} \rangle$; (6.86)
- $e'_{s^j} \sim e_{s^j}$. (6.87)

By the definition of E,

- E is an Y_v-computation. (6.88)

By (6.14), (6.83), and (6.87), $L \circ E$ does not contain any transition events. Moreover, by the definition of L and E, $(L \circ E) \mid p \neq \langle\rangle$ implies $p \in Y_v$, for each process p. Combining these assertions with (6.66), we have

$$\begin{aligned} \mathrm{Act}(\overline{G}) &= \mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = Y_v \quad \wedge \\ \mathrm{Fin}(\overline{G}) &= \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \end{aligned} \tag{6.89}$$

We now state and prove two claims regarding $\overline{G}$. Claim 3 follows easily from the re-indexing of Y_v and construction of E, described above.

Claim 3: Events in E appear in the following order, where α is a fixed value in the range of v and $W(v)$, $C_1(v)$, $C_2(v)$, and $R(v)$ are sets of events.

- events in $W(v)$: each event e'_s in $W(v)$ satisfies $op(e'_s) = \mathsf{write}(v)$;
- events in $C_1(v)$: each event e'_s in $C_1(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \beta_s)$ for some $\beta_s \neq \alpha$;
- events in $C_2(v)$: each event e'_s in $C_2(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \alpha)$;
- events in $R(v)$: each event e'_s in $R(v)$ satisfies $op(e'_s) = \mathsf{read}(v)$.

Moreover, in the computation $\overline{G}$, after all events in $W(v)$ are executed, and before any event in $C_2(v)$ is executed, v has the value α. All events in $C_1(v)$ (if any) are unsuccessful comparisons. At most one event in $C_2(v)$ is a successful comparison. (Note that a successful comparison event writes a value other than α, by definition. Thus, if there is a successful comparison, then all subsequent comparison events must fail.) Define $LW(v)$, the "last write," and $SC(v)$, the "successful comparison," as follows:

$$LW(v) = \begin{cases} \text{the last event in } W(v), & \text{if } W(v) \neq \{\}, \\ \mathit{writer\_event}(v, H' \circ L), & \text{if } W(v) = \{\}; \end{cases}$$

$$SC(v) = \begin{cases} \text{the successful comparison in } C_2(v), & \text{if } C_2(v) \text{ contains one,} \\ \perp, & \text{otherwise.} \end{cases}$$

Then, the last process to write to v (if any) is either $SC(v)$ (if $SC(v)$ is defined) or $LW(v)$ (otherwise). $\square$

Before establishing our next claim, Claim 4, we define p_{LW} and p_{SC} as $\mathit{owner}(LW(v))$ and $\mathit{owner}(SC(v))$, respectively. If $LW(v)$ (respectively, $SC(v)$) equals $\perp$, then p_{LW} (respectively, p_{SC}) also equals $\perp$. We also define RFS as

$$RFS = \mathrm{Fin}(H) \cup \{p\colon p \in \{p_{\mathrm{LW}}, p_{\mathrm{SC}}\} \text{ and } p \neq \perp\}. \tag{6.90}$$

By the definition of Y_v, for each $p \in Y_v$, e_p remotely accesses v. In particular,

- for each $p \in Y_v$, v is remote to p. (6.91)

Note that "expanding" a valid RF-set does not falsify any of RF1–RF5. Therefore, using (6.82), (6.89), and $\mathrm{Fin}(H) \subseteq RFS \subseteq \mathrm{Fin}(H) \cup Y_v$, it follows that

- RFS is a valid RF-set of $H' \circ L$. (6.92)

We now establish Claim 4, stated below.

Claim 4: Every event in E is critical in $\overline{G}$. Also, $\overline{G}$ satisfies RF5.

Proof of Claim: Define $E_0 = \langle\rangle$; for each positive j, define E_j to be $\langle e'_{s^1}, e'_{s^2}, \ldots, e'_{s^j}\rangle$, a prefix of E. We prove the claim by induction on j, applying Lemma 5.2 at each step. Note that, by (6.86) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{s^{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{6.93}$$

Also, by the definition of E_j, we have

$$E_j \mid s^{j+1} = \langle\rangle, \quad \text{for each } j. \tag{6.94}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5. (6.95)

The induction base ($j = 0$) follows easily from (6.92), since $E_0 = \langle\rangle$.

Assume that (6.95) holds for a particular value of j. Since $s^{j+1} \in Y_v \subseteq Y$, we have

$$s^{j+1} \in Y, \tag{6.96}$$

and $s^{j+1} \in \mathrm{Act}(H)$. By applying (6.17) with 'p' $\leftarrow s^{j+1}$, and using (6.96), we also have $\mathrm{Act}(H \circ L_{s^{j+1}}) = \mathrm{Act}(H)$, and hence

$$s^{j+1} \in \mathrm{Act}(H \circ L_{s^{j+1}}). \tag{6.97}$$

Also, by (6.91),

- no events in E_j access any of s^{j+1}'s local variables. (6.98)

We use Lemma 5.2 twice in sequence in order to prove Claim 4. First, by P3, and applying (6.72), (6.81), and (6.85) with 's' $\leftarrow s^{j+1}$, it follows that there exists an event $e''_{s^{j+1}}$, such that

- $H' \circ L \circ \langle e''_{s^{j+1}} \rangle \in C$, and (6.99)
- $e''_{s^{j+1}} \sim e_{s^{j+1}}$. (6.100)

We now apply Lemma 5.2, with 'H' $\leftarrow H \circ L_{s^{j+1}}$, '$H'$' $\leftarrow H' \circ L$, 'G' $\leftarrow \langle\rangle$, '$RFS$' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e_{s^{j+1}}$, and 'e'_p' $\leftarrow e''_{s^{j+1}}$. Among the assumptions stated in Lemma 5.2, (5.5) and (5.7)–(5.9) follow from (6.99), (6.82), (6.100), and (6.97), respectively; (5.11) and (5.12) hold vacuously by 'G' $\leftarrow \langle\rangle$; (5.4), (5.6), and (5.10) follow by applying (6.72), (6.75), and (6.84), respectively, with 's' $\leftarrow s^{j+1}$; (5.13) follows by applying (6.16) with 'p' $\leftarrow s^{j+1}$, and using (6.96). It follows that

- $e''_{s^{j+1}}$ is critical in $H' \circ L \circ \langle e''_{s^{j+1}} \rangle$. (6.101)

Before applying Lemma 5.2 again, we establish the following preliminary assertions. Since $\mathrm{Fin}(H) \subseteq RFS$, by applying (6.75) with 's' $\leftarrow s^{j+1}$, it follows that

- RFS is a valid RF-set of $H \circ L_{s^{j+1}}$. (6.102)

We now establish a simple claim.

> **Claim 4-1:** If $e_{s^{j+1}}$ is a comparison event on v, and if E_j contains a write to v, then $E_j \mid RFS$ also contains a write to v.
>
> **Proof of Claim:** By (6.87) and Claim 3, we have $e'_{s^{j+1}} \in C_1(v) \cup C_2(v)$. Hence, by Claim 3, if an event e'_{s^k} (for some $k \leq j$) in E_j writes to v, then we have either $e'_{s^k} \in W(v)$ or $e'_{s^k} = SC(v)$. If $e'_{s^k} = SC(v)$, then since $s^k \in RFS$ holds by (6.90), Claim 4-1 is satisfied. On the other hand, if $e'_{s^k} \in W(v)$, then $W(v)$ is nonempty. Moreover, since all events in $W(v)$ are indexed before any events in $C_1(v) \cup C_2(v)$, E_j contains all events in $W(v)$. Thus, by (6.90), both E_j and $E_j \mid RFS$ contain $LW(v)$, an event that writes to v. $\square$

We now apply Lemma 5.2 again, with 'H' $\leftarrow H' \circ L$, 'H'' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'e_p' $\leftarrow e''_{s^{j+1}}$, and 'e'_p' $\leftarrow e'_{s^{j+1}}$. Among the assumptions stated in Lemma 5.2, (5.4)–(5.7) and (5.11)–(5.13) follow from (6.99), (6.93), (6.92), (6.92), (6.94), (6.98), and (6.101), respectively; (5.10) is trivial; (5.8) follows from (6.100) and by applying (6.87) with 's^j' $\leftarrow s^{j+1}$; (5.9) follows from (6.89) and $s^{j+1} \in Y_v$. Moreover, Assumption (A) follows from (6.95), and Assumption (B) follows from Claim 4-1.

It follows that $e'_{s^{j+1}}$ is critical in $H' \circ L \circ E_j \circ \langle e'_{s^{j+1}} \rangle = H' \circ L \circ E_{j+1}$, and that $H' \circ L \circ E_{j+1}$ satisfies RF5. $\square$

We now show that RFS is a valid RF-set of $\overline{G}$. Condition RF5 was already proved in Claim 4.

- **RF1 and RF2:** Define E_j as in Claim 4. We establish RF1 and RF2 by induction on j, applying Lemma 5.3 at each step. At each step, we assume

  - $H' \circ L \circ E_j$ satisfies RF1 and RF2. (6.103)

The induction base ($j = 0$) follows easily from (6.92), since $E_0 = \langle\rangle$.

Assume that (6.103) holds for a particular value of j. By Claim 3, if $e'_{s^{j+1}}$ reads v, then the following holds: $e'_{s^{j+1}} \in C_1(v) \cup C_2(v) \cup R(v)$; every event in $W(v)$ is contained in E_j; $\mathit{writer}(v, H' \circ L \circ E_j)$ is one of $\mathit{LW}(v)$ or $\mathit{SC}(v)$ or $\perp$. Therefore, by (6.90), we have the following:

- if $e'_{s^{j+1}}$ remotely reads v, and if we let $q = \mathit{writer}(v, H' \circ L \circ E_j)$, then either $q = \perp$ or $q \in \mathit{RFS}$ holds. (6.104)

We now apply Lemma 5.3, with 'H' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'e_p' $\leftarrow e'_{s^{j+1}}$, and 'v_{rem}' $\leftarrow v$. Among the assumptions stated in Lemma 5.3, (5.14), (5.15), (5.17), (5.19), and (5.21) follow from (6.93), (6.92), (6.103), (6.94), and (6.104), respectively; (5.16) follows from (6.89) and $s^{j+1} \in Y_v$; (5.18) follows from (6.89) and (6.88); (5.20) follows from (6.89) and (6.91). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2.

- **RF3:** Consider a variable $u \in V$ and two different events f_p and g_q in $\overline{G}$. Assume that both p and q are in $\mathrm{Act}(\overline{G})$, $p \neq q$, that there exists a variable u such that $u \in \mathit{var}(f_p) \cup \mathit{var}(g_q)$, and that there exists a write to u in $\overline{G}$. Define $r = \mathit{writer}(u, \overline{G})$. Our proof obligation is to show that $r \in \mathit{RFS}$.

  By (6.89), we have $\{p, q\} \subseteq Y_v$. If $u = v$, then by Claim 3, $\mathit{writer\_event}(u, \overline{G})$ is either $\mathit{SC}(u)$ (if $\mathit{SC}(u) \neq \perp$) or $\mathit{LW}(u)$ (otherwise). (Since we assumed that there exists a write to u, they both cannot be $\perp$.) Thus, by (6.90), we have $r \in \mathit{RFS}$.

  On the other hand, assume $u \neq v$. We now consider three cases.

  - Consider the case in which both f_p and g_q are in $H' \circ L$.

    If there exists an event e'_s in E such that $u \in \mathit{Wvar}(e'_s)$, then since $u \neq v$, u is local to s. Since at least one of p or q is different from s, without loss of generality, assume $p \neq s$. Since $p \in Y_v$ and $Y_v \subseteq \mathrm{Act}(H)$, we have $p \notin \mathrm{Fin}(H)$. Thus, by (6.82) and by applying RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p in $H' \circ L$, we have $s \notin \mathrm{Act}(H' \circ L)$. However, by (6.89), $\mathrm{Act}(H' \circ L) = Y_v$, which contradicts $s \in Y_v$ (which follows from (6.88), since e'_s is an event of E).

    It follows that there exists no event e'_s in E such that $u \in \mathit{Wvar}(e'_s)$ holds. Thus, we have $r = \mathit{writer}(u, H' \circ L)$. By (6.82) and applying RF3 with

'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p and g_q in $H' \circ L$, we have $\mathit{writer}(u, H' \circ L) \in \mathrm{Fin}(H) \subseteq \mathit{RFS}$.

– Consider the case in which f_p is in $H' \circ L$ and $g_q = e'_{s^k}$, for some $s^k \in Y_v$. By (6.89) and our assumption that p and q are both in $\mathrm{Act}(\overline{G})$, we have $p \in \mathrm{Act}(H' \circ L)$ and $q \in \mathrm{Act}(H' \circ L)$. Since $u \neq v$, u is local to q. However, by (6.82), and by applying RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p in $H' \circ L$, we have $q \notin \mathrm{Act}(H' \circ L)$, a contradiction.

– Consider the case in which $f_p = e'_{s^j}$ and $g_q = e'_{s^k}$, for some s^j and s^k in Y_v. Since u is remote to at least one of s^j or s^k, we have $u = v$, a contradiction.

- **RF4:** By (6.14), (6.82), and (6.89), it easily follows that $\overline{G}$ satisfies RF4 with respect to RFS.

Therefore, we have established that

- RFS is a valid RF-set of $\overline{G}$. (6.105)

By (6.89) and (6.90), we have

$$\mathrm{Pmt}_{\mathit{RFS}}(\overline{G}) \; = \; \{p \colon p \in \{p_{\mathrm{LW}}, p_{\mathrm{SC}}\} \text{ and } p \neq \bot\}.$$

In particular,

$$|\mathrm{Pmt}_{\mathit{RFS}}(\overline{G})| \leq 2. \tag{6.106}$$

We now let the processes in $\mathrm{Pmt}(\overline{G})$ finish their execution by inductively appending critical events of processes in $\mathrm{Pmt}(\overline{G})$, thus generating a sequence of computations G_0, G_1, …, G_l (where $G_0 = \overline{G}$), satisfying the following:

- $G_j \in C$; (6.107)
- RFS is a valid RF-set of G_j; (6.108)
- $\mathrm{Pmt}(G_j) \subseteq \mathrm{Pmt}(\overline{G})$; (6.109)
- each process in $\mathrm{Inv}(G_j)$ executes exactly $c+1$ critical events in G_j; (6.110)
- the processes in $\mathrm{Pmt}(\overline{G})$ collectively execute exactly $|\mathrm{Pmt}(\overline{G})| \cdot (c+1) + j$ critical events in G_j; (6.111)
- $\mathrm{Inv}(G_{j+1}) \subseteq \mathrm{Inv}(G_j)$ and $|\mathrm{Inv}(G_{j+1)}| \geq |\mathrm{Inv}(G_j)| - 1$ if $j < l$; (6.112)
- $\mathrm{Fin}(G_j) \subsetneq \mathit{RFS}$ if $j < l$, and $\mathrm{Fin}(G_j) = \mathit{RFS}$ if $j = l$. (6.113)

At each induction step, we apply Lemma 5.6 to G_j in order to construct G_{j+1}, until $\mathrm{Fin}(G_j) = \mathit{RFS}$ is established, at which point the induction is completed. The induction is explained in detail below.

Induction base ($j = 0$): Since $G_0 = \overline{G}$, (6.107) and (6.108) follow from (6.86) and (6.105), respectively. Condition (6.109) is trivial.

By (6.4), (6.69), and (6.83), each process in Y_v executes exactly c critical events in $H' \circ L$. Thus, by Claim 4, it follows that each process in Y_v executes exactly $c + 1$ critical events in $\overline{G}$. Since $\mathrm{Inv}(\overline{G}) \subseteq Y_v$, $\overline{G}$ satisfies (6.110). Since $\mathrm{Pmt}(\overline{G}) \subseteq Y_v$, $\overline{G}$ satisfies (6.111).

Induction step: At each step, we assume (6.107)–(6.111). If $\mathrm{Fin}(G_j) = \mathit{RFS}$, then (6.113) is satisfied and we finish the induction, by letting $l = j$.

Assume otherwise. We apply Lemma 5.6 with 'H' $\leftarrow G_j$. Assumptions (5.35)–(5.37) stated in Lemma 5.6 follow from (6.107), (6.108), and $\mathrm{Fin}(G_j) \neq \mathit{RFS}$. The lemma implies that a computation G_{j+1} exists satisfying (6.107)–(6.113), as shown below.

Condition (6.107) and (6.108) follow from (5.38) and (5.39), respectively. Since G_j satisfies (6.109), by (5.46), G_{j+1} also satisfies (6.109). Since $\mathrm{Inv}(G_{j+1)} \subseteq \mathrm{Inv}(G_j)$ by (5.40) and (5.41), by (5.43) and (5.47), and applying (6.110) to G_j, it follows that G_{j+1} satisfies (6.110). By (5.43)–(5.47), and applying (6.109) and (6.111) to G_j, it follows that G_{j+1} satisfies (6.111). Condition (6.112) follows from (5.40) and (5.41). Thus, the induction is established. □

We now show that $l < 2k$. Assume otherwise. By (6.106), and by applying (6.111) to G_l, it follows that there exists a process $p \in \mathrm{Pmt}(\overline{G})$ (*i.e.*, p is either p_{LW} or p_{SC}) such that p executes at least $c + 1 + k$ critical events in G_l. From (6.113) and $p \in \mathrm{Pmt}(\overline{G}) \subseteq \mathit{RFS}$, we get $p \in \mathrm{Fin}(G_l)$. Let $\overline{F} = G_l \mid \mathit{RFS}$. By Lemma 5.1, and applying (6.107) and (6.108), we have the following:

- $\overline{F} \in C$;
- RFS is a valid RF-set of $\overline{F}$;
- p executes at least $c + 1 + k$ critical events in $\overline{F}$.

Since $p \in \text{Fin}(G_l)$, by applying RF4 to p in G_l, it follows that the last event of $G_l \mid p$ is Exit_p. Since $G_l \mid p = \overline{F} \mid p$, $\overline{F}$ can be written as $F \circ \langle \mathit{Exit}_p \rangle \circ \cdots$, where F is a prefix of $\overline{F}$ such that p executes at least $c + k$ critical events in F. However, p and F then satisfy Proposition Pr1, a contradiction.

Finally, we show that G_l satisfies Proposition Pr2. The following derivation establishes (6.7).

$$\begin{array}{rcll}
|\text{Act}(G_l)| & = & |\text{Inv}_{\mathit{RFS}}(G_l)| & \{\text{by (6.113), } \mathit{RFS} = \text{Fin}(G_l)\text{, thus } \text{Act}(G_l) = \text{Inv}_{\mathit{RFS}}(G_l)\} \\
& \geq & |\text{Inv}_{\mathit{RFS}}(G_0)| - l & \{\text{by repeatedly applying (6.112)}\} \\
& = & |\text{Act}(\overline{G}) - \mathit{RFS}| - l & \{\text{by the definition of ``Inv''; note that } \overline{G} = G_0\} \\
& = & |Y_v - \mathit{RFS}| - l & \{\text{by (6.89)}\} \\
& = & |Y_v - (\text{Pmt}(\overline{G}) \cup \text{Fin}(H))| - l & \{\text{because } \mathit{RFS} = \text{Pmt}(\overline{G}) \cup \text{Fin}(\overline{G})\text{, and} \\
& & & \quad \text{Fin}(\overline{G}) = \text{Fin}(H) \text{ by (6.89)}\} \\
& = & |Y_v - \text{Pmt}(\overline{G})| - l & \{\text{because } Y_v \cap \text{Fin}(H) = \{\} \text{ by (6.89)}\} \\
& > & |Y_v| - 2 - 2k & \{\text{by (6.106) and } l < 2k\} \\
& > & \sqrt{n} - 2k - 3. & \{\text{by (6.64)}\}
\end{array}$$

Moreover, by (6.108) and (6.113), we have $\text{Act}(G_l) = \text{Inv}(G_l)$. Thus, by (6.89) and (6.112), we have $\text{Act}(G_l) \subseteq \text{Inv}(\overline{G}) \subseteq \text{Act}(\overline{G}) = Y_v \subseteq \text{Act}(H)$, which implies (6.5). By (6.90) and (6.113), we have (6.6). Finally, (6.110) implies (6.8). Therefore, G_l satisfies Proposition Pr2. $\square$

Theorem 6.1 *Let $\bar{N}(k) = (2k + 4)^{2(2^k - 1)}$. For any mutual exclusion system $\mathcal{S} = (C, P, V)$ and for any positive number k, if $|P| \geq \bar{N}(k)$, then there exists a computation H such that at most $2k - 1$ processes participate in H and some process p executes at least k critical operations in H to enter and exit its critical section.*

Proof: Let $H_1 = \langle \mathit{Enter}_1,\ \mathit{Enter}_2,\ \ldots,\ \mathit{Enter}_N \rangle$, where $P = \{1, 2, \ldots, N\}$ and $N \geq \bar{N}(k)$. By the definition of a mutual exclusion system, $H_1 \in C$. It is obvious that H_1 is regular and each process in $\text{Act}(H) = P$ has exactly one critical event in H_1. Starting with H_1, we repeatedly apply Lemma 6.1 and construct a sequence of computations $(H_1,\ H_2,\ \ldots)$, such that each process in $\text{Act}(H_j)$ has j critical events in H_j. We repeat the process until either H_k is constructed or some H_j satisfies Proposition Pr1 of Lemma 6.1.

If some H_j $(j \leq k-1)$ satisfies Proposition Pr1, then consider the first such j. By our choice of j, each of $H_1, \ \ldots, \ H_j - 1$ satisfies Proposition Pr2 of Lemma 6.1. Therefore, since $|\mathrm{Fin}(H_1)| = 0$, we have $|\mathrm{Fin}(H_j)| \leq 2(j-1) \leq 2k-4$. It follows that computation $F \circ \langle \mathit{Exit}_p \rangle$, generated by applying Lemma 6.1 to H_j, satisfies Theorem 6.1.

The remaining possibility is that each of $H_1, \ \ldots, \ H_{k-1}$ satisfies Proposition Pr2. We claim that, for $1 \leq j \leq k$, the following holds:

$$|\mathrm{Act}(H_j)| \geq (2k+4)^{2(2^{k+1-j}-1)}. \tag{6.114}$$

The induction basis $(j = 1)$ directly follows from $\mathrm{Act}(H) = P$ and $|P| \geq \bar{N}(k)$. In the induction step, assume that (6.114) holds for some j $(1 \leq j < k)$, and let $n_j = |\mathrm{Act}(H_j)|$. Note that each active process in H_j executes exactly j critical events. By (6.114), we also have $n_j > (2k+4)^2$, which in turn implies that $\sqrt{n_j} - 2k - 3 > \sqrt{n_j}/(2k+4)$. Therefore, by (6.7), we have

$$|\mathrm{Act}(H_{j+1)}| \ \geq \ \min(\sqrt{n_j}/(2j+3), \ \sqrt{n_j} - 2k - 3) \ \geq \ \sqrt{n_j}/(2k+4),$$

from which the induction easily follows.

Finally, (6.114) implies $|\mathrm{Act}(H_k)| \geq 1$, and Proposition Pr2 implies $|\mathrm{Fin}(H_k)| \leq 2(k-1)$. Therefore, select any arbitrary process p from $\mathrm{Act}(H_k)$. Define $G = H_k \mid (\mathrm{Fin}(H_k) \cup \{p\})$. Clearly, at most $2k-1$ processes participate in G. By applying Lemma 5.1 with 'H' $\leftarrow H_k$ and 'Y' $\leftarrow \mathrm{Fin}(H_k) \cup \{p\}$, we have the following: $G \in C$, and an event in G is critical if and only if it is also critical in H_k. Hence, because p executes k critical events in H_k, G is a computation that satisfies Theorem 6.1. □

6.3 Concluding Remarks

We have established a lower bound that eliminates the possibility of an $O(\log k)$ adaptive mutual exclusion algorithm based on reads, writes, or comparison primitives, where k is either point or interval contention.

We believe that $\Omega(\min(k, \log N))$ is probably a tight lower bound for the class of algorithms considered in this chapter (which would imply that ALGORITHM A-LS is optimal). One relevant question is whether the results of this chapter can be combined with those presented in Chapter 5 to come close to an $\Omega(\min(k, \log N))$ bound, *i.e.*, can we at least conclude that $\Omega(\min(k, \log N/\log\log N))$ is a lower bound? Unfortunately, the answer is no. We have shown that $\Omega(k)$ RMR time complexity is required *provided* N is sufficiently large. This leaves open the possibility that an algorithm might have $\Theta(k)$ RMR time complexity for very "low" levels of contention, but $o(k)$ RMR time complexity for "intermediate" levels of contention. Although our lower bound does not preclude such a possibility, we find it highly unlikely.

CHAPTER 7

Algorithm and Time-complexity Lower Bound for Nonatomic Systems*

In this chapter, we present an N-process local-spin mutual exclusion algorithm with $\Theta(\log N)$ RMR (remote-memory-reference) time complexity, which is based on nonatomic reads and writes. This algorithm is derived from Yang and Anderson's atomic local-spin algorithm (ALGORITHM YA-N), which was described in Sections 2.2.2 and 3.1, in a way that preserves its time complexity. No atomic read/write algorithm with better asymptotic worst-case time complexity (under the RMR measure) is currently known. This suggests that atomic memory is *not* fundamentally required if one is interested in *worst-case* time complexity.

The same cannot be said if one is interested in fast or adaptive algorithms. We show that such algorithms fundamentally require memory accesses to be atomic. In particular, we show that for any N-process nonatomic algorithm, there exists a single-process execution in which the lone competing process accesses $\Omega(\log N/\log\log N)$ distinct variables to enter its critical section. Thus, fast and adaptive algorithms are impossible even if caching techniques are used to avoid accessing the processors-to-memory interconnection network.

*The results presented in this chapter have been published in the following paper.
[18] J. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 3–12. ACM, July 2002.

In nonatomic algorithms, variable accesses[1] are assumed to take place over intervals of time, and hence may overlap one another. In contrast, each variable access in an atomic algorithm is viewed as taking place instantaneously. Requiring atomic memory access is tantamount to assuming mutual exclusion in hardware [51]. Thus, mutual exclusion algorithms requiring this are in some sense circular.

As discussed in Section 2.5, Anderson devised the first nonatomic local-spin mutual exclusion algorithm [11]. This algorithm is composed of a collection of constant-time two-process algorithms, which are used to allow each process to compete individually against every other process. The resulting algorithm has $\Theta(N)$ RMR time complexity, where N is the number of processes. The correctness of the nonatomic version of the algorithm is mainly a consequence of the fact that only single-writer, single-reader, single-bit variables are used. With any nonatomic algorithm, overlapping operations that access a common variable are the main concern. In Anderson's algorithm, if two overlapping operations access the same (single-bit) variable, then one is a read and the other is a write. The assumption usually made (and made herein) regarding such overlapping operations is that the read may return any value [51, 55]. Note that if such a write changes the written variable's value, then an overlapping read can be linearized to occur either before or after the write [51]. For example, if a write changes a variable's value from 1 to 0, then an overlapping read that returns 1 (0) can be linearized to occur immediately before (after) the write.[2] In Anderson's algorithm, most writes write new values, and the structure of the algorithm ensures that those writes that do not have no adverse impact.

In recent years, there has been much interest in fast and adaptive algorithms (see Sections 2.3 and 2.4). In a recent paper [16], we presented a "fast-path" mechanism that improves the contention-free time complexity of Yang and Anderson's algorithm to $O(1)$, without affecting its worst-case time complexity. In Chapter 4, we have presented an extension of this mechanism that results in an adaptive algorithm with $O(\min(k, \log N))$ RMR time complexity, where k is point contention.

As described in Section 2.6, Anderson and Yang [22] established trade-offs between time complexity and write- and access-contention for mutual exclusion algorithms (Theorems 2.10, 2.11). The *write-contention* (*access-contention*) of a concurrent program

[1]In this chapter, we only consider algorithms based on reads and writes, unless otherwise indicated.

[2]Such reasoning must be used with caution; for example, it may be impossible to linearize a *sequence* of such overlapping reads by the same process without reordering them.

is the number of processes that may be simultaneously enabled to write (access by reading and/or writing) the same shared variable.

Anderson and Yang showed that any algorithm with write-contention w must have a single-process execution in which that process executes $\Omega(\log_w N)$ remote operations for entry into its critical section.[3] Further, among these operations, $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed. Similarly, any algorithm with access-contention c must have a single-process execution in which that process accesses $\Omega(\log_c N)$ distinct remote variables for entry into its critical section. Thus, a trade-off between write-contention (access-contention) and time complexity exists even in systems with coherent caches.

Because a single-process execution is used to establish these bounds, it follows that $\Omega(N^\epsilon)$-writer variables (for some positive constant ϵ) are needed for fast or adaptive algorithms. In other work, Alur and Taubenfeld [9] showed that fast (and hence adaptive) algorithms also require variables with $\Omega(\log N)$ bits (*i.e.*, variables large enough to hold at least some fraction of a process identifier). (See Theorem 2.13.)

In Chapter 5, we established a lower bound of $\Omega(\log N/\log\log N)$ remote operations for any mutual exclusion algorithm based on reads and writes. This bound has no bearing on fast or adaptive algorithms because it results from an execution that may involve many processes. (In Chapter 6, adaptive *atomic* algorithms were considered.)

Given the research reviewed above, two questions immediately come to mind:

- Is it possible to devise a nonatomic local-spin algorithm with $\Theta(\log N)$ time complexity, *i.e.*, that matches the best atomic algorithm known?

- Is it possible to devise a nonatomic algorithm that is fast or adaptive?

Both questions are answered in this chapter. We answer the first question in the affirmative by presenting a $\Theta(\log N)$ nonatomic algorithm, which is derived from ALGORITHM YA-N by means of simple transformations. On the other hand, the answer to the second question is negative. We show this by proving that any nonatomic algorithm must have a single-process execution in which that process accesses $\Omega(\log N/\log\log N)$ distinct variables. Therefore, fast and adaptive algorithms are impossible even if caching techniques are used to avoid accessing the interconnection network. Given the prior results summarized above, it follows that any fast or adaptive algorithm necessarily must use $\Omega(\log N)$-bit, $\Omega(N^\epsilon)$-writer variables that are accessed *atomically*.

[3] Recall that we only consider algorithms based on reads and writes in this chapter. Thus, the parameter v stated in Theorems 2.10 and 2.11 equals one.

This chapter is organized as follows. In Section 7.1, our nonatomic algorithm is presented; a correctness proof for it is given in Appendix C. Definitions needed to establish the above-mentioned lower bound are then given in Section 7.2. The lower-bound proof is sketched in Section 7.3; a full proof is given in Appendix D. We conclude in Section 7.4.

7.1 Nonatomic Algorithm

As mentioned earlier, our nonatomic algorithm, denoted as Algorithm NA hereafter, is derived from Algorithm YA-N. We illustrate both algorithms in Figure 7.1. For a detailed description of Algorithm YA-N, we refer the reader to Section 3.1.

We now consider the problem of converting Algorithm YA-N into a nonatomic algorithm. The notion of a nonatomic variable that we assume is that captured by Lamport's definition of a *safe register* [55]: a nonatomic read of a variable returns its current value if it does not overlap any write of that variable, and any arbitrary value from the value domain of the variable if it does overlap such a write. These assumptions are sufficient for our purposes, because our final algorithm precludes overlapping writes of the same variable.

The most obvious way to convert Algorithm YA-N into a nonatomic algorithm is to implement each atomic variable using nonatomic ones by applying wait-free register constructions presented previously [39, 40, 55, 70, 74, 77]. This is in fact the approach we take for the C and T variables. However, if such constructions are applied to implement the Q variables, then a read of such a variable necessarily requires that one or more of the underlying nonatomic variables be written. (This was proved by Lamport [55].) As a result, the spins in statements 9 and 11 would no longer be local.

As for the C and T variables, the tree structure ensures that $C[n][s]$ can be viewed as a single-writer, single-reader variable, and $T[n]$ as a two-writer, two-reader variable. Hence, $C[n][s]$ can be implemented quite efficiently using the single-writer, single-reader register construction of Haldar and Subramanian [39]. In this construction, eight nonatomic variables are used, each atomic read requires at most four accesses of nonatomic variables, and each atomic write at most seven. $T[n]$ is more problematic, as it is a multi-reader, multi-writer atomic variable. Nonetheless, register constructions are known that can be used to implement such variables from nonatomic variables with time and space complexity that is polynomial in the number of readers and writers [40, 55, 70, 74, 77]. For variable $T[n]$, the number of readers and writers is constant.

ALGORITHM YA-N (The original algorithm in [84])

```
process $p$ ::    /* $0 \le p < N$ */
const /* for simplicity, we assume $N = 2^L$ */
    $L = \log N$;
                /* (tree depth) + 1 = $O(\log N)$ */
    $Tsize = 2^L - 1 = N - 1$
                                /* tree size = $O(N)$ */
shared variables
    $T$: array[1..$Tsize$] of $0..N-1$;
    $C$ : array[1..$Tsize$][0..1] of $(0..N-1,\ \bot)$
        initially $\bot$;
    $Q$: array[1..$L$][0..$N-1$] of 0..2
        initially 0
private variables
    $h$: 1..$L$;
    $node$: 1..$Tsize$;
    $side$: 0..1;              /* 0 = left, 1 = right */
    $rival$: $0..N-1$

while $true$ do
1:  Noncritical Section;
    for $h := 1$ to $L$ do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
2:      $C[node][side] := p$;
3:      $T[node] := p$;
4:      $Q[h][p] := 0$;
5:      $rival := C[node][1 - side]$;
6:      if $(rival \neq \bot \ \wedge\ T[node] = p)$ then
7:          if $Q[h][rival] = 0$ then
8:              $Q[h][rival] := 1$ fi;
9:          await $Q[h][p] \geq 1$;
10:         if $T[node] = p$ then
11:             await $Q[h][p] = 2$ fi
        fi
    od;
12: Critical Section;
    for $h := L$ downto 1 do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
13:     $C[node][side] := \bot$;
14:     $rival := T[node]$;
        if $rival \neq p$ then
15:         $Q[h][rival] := 2$ fi
    od
od
```

(a)

ALGORITHM NA (New nonatomic algorithm)

```
process $p$ ::    /* $0 \le p < N$ */
/* all variable declarations are as in */
/* ALGORITHM YA-N, except that */
/* $P$ is replaced by the following    */
shared variables
    $Q1, Q2, R1, R2$:
        array[1..$L$][0..$N-1$] of boolean
private variables
    $qtoggle$, $rtoggle$, $temp$: 0..1

while $true$ do
1:  Noncritical Section;
    for $h := 1$ to $L$ do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
2:      $C[node][side] := p$;
3:      $T[node] := p$;
4:      $rtoggle := \neg R1[h][p]$;
5:      $R2[h][p] := rtoggle$;
6:      $qtoggle := \neg Q1[h][p]$;
7:      $Q2[h][p] := qtoggle$;
8:      $rival := C[node][1 - side]$;
9:      if $(rival \neq \bot \ \wedge\ T[node] = p)$ then
10:         $temp := Q2[h][rival]$;
11:         $Q1[h][rival] := temp$;
12:         await $(Q1[h][p] = qtoggle) \vee$
13:                 $(R1[h][p] = rtoggle)$;
14:         if $T[node] = p$ then
15:             await $R1[h][p] = rtoggle$ fi
        fi
    od;
16: Critical Section;
    for $h := L$ downto 1 do
        $node := \lfloor (N+p)/2^h \rfloor$;
        $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
17:     $C[node][side] := \bot$;
18:     $rival := T[node]$;
        if $rival \neq p$ then
19:         $temp := R2[h][rival]$;
20:         $R1[h][rival] := temp$ fi
    od
od
```

(b)

Figure 7.1: **(a)** ALGORITHM YA-N and **(b)** its nonatomic variant. In (b), reads and writes of the C and T variables are assumed to be implemented using register constructions.

Thus, it can be implemented using nonatomic variables with constant space and time complexity.

The need for register constructions to implement the T variables can be obviated by slightly modifying the algorithm, using a technique first proposed by Kessels [45]. (For ease of exposition, this is not done in ALGORITHM NA in Figure 7.1(b).) The idea is to replace each $T[n]$ variable by two single-bit variables $T1[n]$ and $T2[n]$; $T1[n]$ ($T2[n]$) is written by left-side (right-side) processes and read by right-side (left-side) processes at node n. Left-side processes seek to establish $T1[n] = T2[n]$ and right-side processes seek to establish $T1[n] \neq T2[n]$. Ties are broken accordingly. Because $T1[n]$ and $T2[n]$ are both single-writer, single-reader, single-bit variables, it is relatively straightforward to show that this mechanism still works if variable accesses are nonatomic. In fact, this very mechanism is used in the nonatomic algorithm of Anderson [11], which was described in Section 2.5.

The Q variables can be dealt with similarly. In ALGORITHM YA-N, the condition $Q[h][p] \geq 1$ indicates that process p may proceed past its first **await**, and the condition $Q[h][p] = 2$ indicates that p may proceed past its second **await**. Because multi-bit variables are problematic if memory accesses are nonatomic, we implement these conditions using separate variables. In ALGORITHM NA (see Figure 7.1(b)), variables $Q1[h][p]$ and $Q2[h][p]$ are used to implement the first condition, and variables $R1[h][p]$ and $R2[h][p]$ are used to implement the second. The technique used in updating both pairs of variables is similar to that used in Kessels' tie-breaking scheme described above. In particular, process p attempts to establish $Q1[h][p] \neq Q2[h][p] \ \wedge \ R1[h][p] \neq R2[h][p]$ in statements 4–7 and waits while this condition continues to hold at statements 12–13 (note that $qtoggle = Q2[h][p] \ \wedge \ rtoggle = R2[h][p]$ holds while p continues to wait).[4] A rival process at node n seeks to establish $Q1[h][p] = Q2[h][p]$ at statements 10–11; the effect is similar to statements 7–8 in ALGORITHM YA-N. Statements 15 and 19–20 work in a similar way. As with Kessels' tie-breaking scheme, because the new variables being used here are all single-writer, single-reader, single-bit variables, it is relatively straightforward to show that the algorithm is correct even if variable accesses are nonatomic. A complete correctness proof for the algorithm is given in Appendix C. This gives us the following theorem.

Theorem 7.1 *The mutual exclusion problem can be solved with* $\Theta(\log N)$ *RMR time complexity using only nonatomic reads and writes.* □

[4]In the preliminary version of the work presented in this chapter [18], statements 6–7 in Figure 7.1(b) precede statements 4–5, writing $Q2$ before $R2$. We found that that version is susceptible to livelock.

7.2 Nonatomic Shared-memory Systems

In this section, we present the model of a nonatomic shared-memory system that is used in our lower-bound proof. Our model of a nonatomic system is similar to that of an atomic system, given in Section 5.1. However, there are two key differences. First, our model of a nonatomic shared-memory system does not include comparison primitives or events that access multiple local (shared) variables, and hence the definition of an event is greatly simplified. Second, we introduce two classes of events, namely *invocation* and *response* events, in order to represent an execution of a nonatomic write. (As explained shortly, we assume that all reads execute fast enough to be considered atomic.)

Nonatomic shared-memory systems. A *nonatomic shared-memory system* $\mathcal{S} = (C, P, V)$ consists of a set of (nonatomic) computations C, a set of processes P, and a set of shared variables V. A *computation* is a finite sequence of events. As in Section 5.1, an *initial value* is associated with each shared variable. In practice, accesses of private variables such as program counters usually determine the order in which shared variables are accessed. For our purposes, the manner in which this access order is determined is not important. Thus, we do not consider private variables nor events that access them in our proof.

In all computations considered in our proof, reads execute atomically (*i.e.*, have zero duration). Writes may execute atomically or nonatomically, but writes to the same variable never overlap each other. Thus, we have no need to define the effects of concurrent writes. As stated earlier, the notion of a nonatomic variable that we assume is that captured by a *safe register* [55]: a nonatomic read of a variable returns its current value if it does not overlap any write of that variable, and any arbitrary value from the value domain of the variable if it does overlap such a write.

We now formally state the definition of an event. An *event* e has the form of $[p, \mathsf{Op}, \ldots]$, where $p \in P$. (The various forms of an event are given in the Atomicity property below.) As in Section 5.1, we call Op the *operation* of event e, denoted $op(e)$. Op can be one of the following: $\mathsf{read}(v)$, $\mathsf{write}(v)$, $\mathsf{invoke}(v)$, or $\mathsf{respond}(v)$, where v is a variable in V. (Informally, e can be an atomic read, an atomic write, an invocation of a nonatomic write, or a corresponding response of a nonatomic write.) As before, we sometimes use e_p to denote an event owned by process p. The following assumption formalizes requirements regarding the atomicity of events.

Atomicity property: Each event e_p must be of one of the forms below.

- $e_p = [p, \mathsf{read}(v), \alpha]$. In this case, e_p reads the value α from v. We call e_p a *read event.*

- $e_p = [p, \mathsf{write}(v), \alpha]$. In this case, e_p writes the value α to v. We call e_p an *atomic write event.*

- $e_p = [p, \mathsf{invoke}(v), \alpha]$. In this case, e_p writes the value $\star$ to v, where $\star$ is a special value that means subsequent reads of v may read any value. We call e_p an *invocation event.*

- $e_p = [p, \mathsf{respond}(v), \alpha]$. In this case, e_p writes the value α to v. We call e_p a *response event.* $\square$

We say that an event e_p *writes* v if $op(e_p) \in \{\mathsf{write}(v),\ \mathsf{invoke}(v),\ \mathsf{respond}(v)\}$, and that e_p *reads* v if $op(e_p) = \mathsf{read}(v)$. We say that e_p *accesses* v if it writes or reads v. We also say that a computation H *contains a write* (respectively, *read*) of v if H contains some event that writes (respectively, reads) v.

An atomic write is merely a notational convenience to represent a write that is executed "fast enough" to be considered atomic. Therefore, we require that a process has an enabled atomic write if and only if it has an identical enabled nonatomic write (Property P6, given later). A nonatomic write is represented by two successive events, for its beginning (invocation) and its end (response). An example is shown in Figure 7.2(a). If a process has performed an invocation event, then it may execute the matching response event (Property P7). As stated before, our lower-bound proof in Chapter 7 ensures that between a matching invocation and response, no write to the same variable ever occurs. Therefore, in order to simplify bookkeeping, we allow response events to be *implicit.* To be precise, if a process p executes an invocation event e_p, and if another process q executes an event f_q that writes v after e_p (but before its matching response event), then the matching response event implicitly occurs immediately before f_q, as shown in Figure 7.2(b). Therefore, in our model, overlapping writes to the same variable cannot happen.[5] (In fact, explicit response events are not used at all in our proof. Although this may lead to an "open" nonatomic write that does not terminate, such a write can always be converted into a "proper" nonatomic write by appending a corresponding explicit response event. Explicit response events

[5]This does not mean that our lower bound does not apply to systems that allow overlapping writes to the same variable. Such a system still has a subset of valid computations in which overlapping writes to the same variable do not happen.

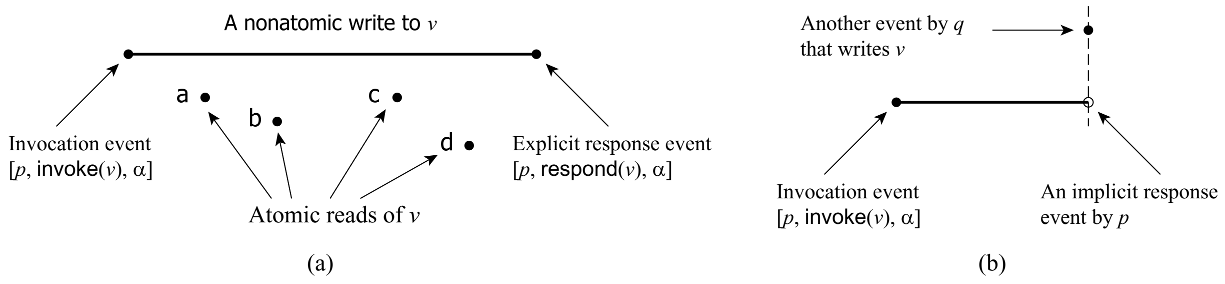

A nonatomic write to v
a
b
c
d
Invocation event [p, invoke(v), α]
Explicit response event [p, respond(v), α]
Atomic reads of v
(a)
Another event by q that writes v
Invocation event [p, invoke(v), α]
An implicit response event by p
(b)

Figure 7.2: Overlapping reads and writes of the same variable. **(a)** A nonatomic write to v by p, terminated by an explicit response event. **(b)** A nonatomic write to v by p, terminated by another process q's write to v. In this case, q's event may be an atomic write event or an invocation event.

are introduced in this section merely to make our system model easier to understand.) Thus, as far as overlapping operations are concerned, the only interesting case is that of a read of a variable overlapping a write of the same variable. In this case, the read may return any value (Properties P2′ and P4′ below). For example, in Figure 7.2(a), events a–d may read any value from v.

Notational conventions pertaining to computations. The definitions of $value(v, H)$, $writer\_event(v, H)$, $writer(v, H)$ are essentially the same as those given in Section 5.1. (Formally, let e_p be the last event in H with operation write(v), invoke(v), or respond(v). We define $writer\_event(v, H) = e_p$, $writer(v, H) = p$, and $value(v, H)$ to be the value written to v by e_p. If no such e_p exists, then we define $writer\_event(v, H) = writer(v, H) = \perp$, and $value(v, H)$ to be the initial value of v.) Other definitions pertaining to atomic computations (given on page 92) also apply to nonatomic computations.

Properties of nonatomic shared-memory systems. The following properties apply to any nonatomic shared-memory system. Note that Property P1, given below, is identical to that given in Section 5.1, while Properties P2′ and P4′ are modified from P2 and P4 in order to reflect the nonatomic nature of our system model. Properties P6 and P7 are exclusive to nonatomic systems.

P1: If $H \in C$ and G is a prefix of H, then $G \in C$.

— *Informally, every prefix of a valid computation is also a valid computation.*

P2′: Assume that $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \mid p = H \mid p$. Also assume that either **(i)** e_p is not a read event, or **(ii)** e_p reads v and $\mathit{value}(v, G) \in \{\mathit{value}(v, H), \star\}$ holds. Then, $G \circ \langle e_p \rangle \in C$ holds.

— *Informally, if two computations H and G are not distinguishable to process p, if p can execute event e_p after H, and if a variable read by e_p (if any), after G, has either the value $\star$ or the same value as after H, then p can execute e_p after G. Note that e_p may read any value from a variable with value $\star$, i.e., a variable that is concurrently being accessed by a nonatomic write.*

P4′: For any $H \in C$ and a read event e_p of v, $H \circ \langle e_p \rangle \in C$ implies that $\mathit{value}(v, H)$ is either α or $\star$.

— *Informally, only the last value written to a variable may be read, unless the last write was an invocation event on that variable.*

P6: For any $H \in C$, $p \in P$, $v \in V$, and α, $H \circ \langle [p, \mathsf{write}(v), \alpha] \rangle \in C$ holds if and only if $H \circ \langle [p, \mathsf{invoke}(v), \alpha] \rangle \in C$ holds.

— *Informally, p can write to v atomically (via $[p, \mathsf{write}(v), \alpha]$) if and only if p can start writing to v nonatomically (via $[p, \mathsf{invoke}(v), \alpha]$).*

P7: For any $H \in C$, $p \in P$, $v \in V$, and α, $H \circ \langle [p, \mathsf{respond}(v), \alpha] \rangle \in C$ holds if and only if $\mathit{writer\_event}(v, H) = [p, \mathsf{invoke}(v), \alpha]$ holds.

— *Informally, a response event on v may appear only if preceded by the corresponding invocation event, and only if there is no intervening write to v, since such a write would entail an implicit response event.*

As stated above, our proof does not make use of explicit response events. Therefore, in the rest of this chapter, we assume that every computation of concern is free of explicit response events.

Nonatomic one-shot mutual exclusion systems. We now define a special kind of nonatomic shared-memory system, namely (nonatomic) one-shot mutual exclusion systems, which are our main interest. Such systems solve a simplified version of the mutual exclusion problem in which the first process that enters its critical section halts immediately. Clearly, a lower bound for one-shot mutual exclusion also implies a lower bound for general mutual exclusion.

A *nonatomic one-shot mutual exclusion system* $\mathcal{S} = (C, P, V)$ is a nonatomic shared-memory system that satisfies the following properties. Each process $p \in P$ has two dummy auxiliary variables, $entry_p$ and cs_p. These variables are accessed only by the following events: $Enter_p = [p, \mathsf{write}(entry_p), 0]$, and $CS_p = [p, \mathsf{write}(cs_p), 0]$.

These events are called *transition events*, and are allowed in the following situations. For all $H \in C$,

- if $H \mid p \neq \langle\rangle$, then the first event (and only the first event) by p in H is $Enter_p$;
- if $H \mid p = \langle\rangle$, then $H \circ \langle Enter_p \rangle \in C$;
- if H contains CS_p, then it is the last event of $H \mid p$.

We say that a process p is in its *entry section* if it has executed $Enter_p$ but not CS_p, and that p is in its *critical section* if it has executed both $Enter_p$ and CS_p. (Processes of a one-shot mutual exclusion system do not have exit sections.) The remaining requirements of a one-shot mutual exclusion system are as follows.

Exclusion: For all $H \in C$, if H contains CS_p, then it does not contain CS_q for any $q \neq p$.

Progress (of a solo computation): For all $H \in C$, if H is a p-computation and H does not contain CS_p, then there exists a p-computation G such that $H \circ G \circ \langle CS_p \rangle \in C$ holds.

Note that the Progress property above is much weaker than that usually specified for the mutual exclusion problem. Clearly, it is satisfied by any livelock-free mutual exclusion algorithm.

Critical events in nonatomic systems. Since our nonatomic lower bound in Chapter 7 is concerned with the number of distinct variables accessed, in order to facilitate the proof, we define certain events as *critical events*. (Since we only have to count the number of variables, the definition given here is much simpler than that of Section 5.1.) An event of p in a computation H is *critical* if it is the first read of some variable v by p or the first atomic write to v by p. Thus, if p has m critical events in H, then it accesses at least $m/2$ distinct variables. Note that $Enter_p$ and CS_p are critical events in any computation, since they appear at most once and access new variables ($entry_p$ and cs_p, respectively).

Consider the solo computation S_p by a process p, in which every write is atomic. As defined above, the first event in S_p must be $Enter_p$. By the Progress property, p

```
write $\boldsymbol{entry_p}$;    /* $Enter_p$ */
write $\boldsymbol{u}$;
read $\boldsymbol{v}$;
read $\boldsymbol{u}$;
read $v$;
write $u$;
write $\boldsymbol{w}$;
write $\boldsymbol{cs_p}$        /* $CS_p$ */
halt
```

```
$S(p,1)$:  write $\boldsymbol{entry_p}$;    /* $Enter_p$ */
$S(p,2)$:  write $\boldsymbol{u}$;
$S(p,3)$:  read $\boldsymbol{v}$;
$S(p,4)$:  read $\boldsymbol{u}$; read $v$; write $u$;
$S(p,5)$:  write $\boldsymbol{w}$;
$S(p,6)$:  write $\boldsymbol{cs_p}$        /* $CS_p$ */
halt
```

Figure 7.3: A possible solo computation by a process p. (a) Critical and noncritical events. Critical events are shown in **boldface**. u, v, and w denote shared variables. Private variable accesses are ignored and are not shown. (b) The same computation, partitioned into solo-execution segments.

eventually executes CS_p, and then terminates. (An example is shown in Figure 7.3(a).) We define $ce(p, j)$ as the j^{th} critical event by p in its solo computation. For example, in Figure 7.3(a), we have $ce(p, 1) = Enter_p$, $ce(p, 2) = [p, \mathsf{write}(u), \alpha]$ (for some α), $ce(p, 3) = [p, \mathsf{read}(v), \beta]$ (for some β), and so on. Note that, for any process p, its first and last critical events are $Enter_p$ and CS_p, respectively.

If $ce(p, j)$ is a critical write event of v, then we also denote its corresponding invocation event on v by $ie(p, j)$. For example, in Figure 7.3, we have $ie(p, 2) = [p, \mathsf{invoke}(u), \alpha]$ (for some α) and $ie(p, 5) = [p, \mathsf{invoke}(w), \gamma]$ (for some γ), but $ie(p, 3)$ is undefined.

We partition S_p into *solo segments* $S(p, j)$ (for $j = 1,\ 2,\ \ldots$), such that $S(p, j)$ starts with p's j^{th} critical event $ce(p, j)$ and ends right before p's $(j+1)^{\text{st}}$ critical event. We say that a solo segment by p is a *transition* segment if it contains $Enter_p$ or CS_p (*i.e.*, if it is either p's first or last solo segment). For example, the solo computation by p in Figure 7.3(b) is divided into six segments, where $S(p, 1)$ and $S(p, 6)$ are transition segments and the rest are non-transition segments.

7.3 Lower Bound: Proof Sketch

In Appendix D, we show that for any one-shot mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation H such that some process p accesses $\Omega(\log N/\log\log N)$ distinct variables to enter its critical section, where $N = |P|$. In this section, we sketch the key ideas of our proof.

7.3.1 Brief Overview

As in Chapters 5 and 6, our proof focuses on a special class of computations called "regular" computations. However, the definition of a regular computation is different from these chapters, because the underlying system model is different.

A regular computation H consists of events of two groups of processes, "active processes" (denoted by $\mathrm{Act}(H)$) and "covering processes" (denoted by $\mathrm{Cvr}(H)$). Informally, an active process is a process in its entry section, competing with other active processes; a covering process is a process that has executed some part of its entry section, and has started (or is ready to start) a nonatomic write (by executing an invocation event) of some variable v in order to "cover" v, so that other processes may concurrently access v without gaining knowledge of each other.

At the end of this section, a detailed overview of our proof is given. Here, we give cursory overview, so that the definitions that follow will make sense. Initially, we start with a regular computation H_1 in which all the processes in P are active and execute their first solo segments (*i.e.*, $\langle \mathit{Enter}_p \rangle$ for each $p \in P$). The proof proceeds by inductively constructing longer and longer regular computations, until the desired lower bound is attained. The regularity condition defined below ensures that no participating process has "knowledge" of any other process that is active.[6] This has three consequences: each process executes the same sequence of events as its solo computation (*i.e.*, as it does when executed alone); we can "erase" any active process (*i.e.*, remove its events from the computation) and still get a valid computation; "most" active processes have a "next" non-transition critical event, and hence, a "next" non-transition solo segment. In each induction step, we append to each of the n active processes (except at most one) its next solo segment. These next solo segments may introduce unwanted information flow, *i.e.*, they may cause an active process to acquire knowledge of another active process, resulting in a non-regular computation. Such information flow is problematic because we are ultimately interested in solo computations.

Information flow among processes is prevented either by covering variables, as described above, or by erasing processes — when a process is erased, its events are completely removed from the computation currently being considered. These basic techniques, covering and erasing, have been previously used to prove other lower bounds pertaining to concurrent systems ([3, 79]; also see Section 2.6). The erasing strategy

[6]A process p has knowledge of other processes if it has read a variable with a value (different from $\star$) written by another process. If a process p reads a variable with the value $\star$, then any value can be returned. We assume the value returned is the same value as in its solo computation.

$$\underbrace{\begin{matrix} S(p_1,1) & & & \\ & S(p_2,1) & & \\ & & \ddots & \\ & & & S(p_k,1) \end{matrix}}_{H^1\text{: added at the first step}} \quad \underbrace{\begin{matrix} S(p_1,2) & & & \\ & S(p_2,2) & & \\ & & \ddots & \\ & & & S(p_k,2) \end{matrix}}_{H^2\text{: added at the second step}} \quad \cdots \quad \underbrace{\begin{matrix} S(p_1,m-1) & & & \\ & S(p_2,m-1) & & \\ & & \ddots & \\ & & & S(p_k,m-1) \end{matrix}}_{H^{m-1}\text{: added at the last } ((m-1)^{\text{st}}) \text{ step}}$$

Figure 7.4: Structure of a regular computation H after $m-1$ induction steps. This diagram does not show the full structure of H; additional details will be introduced as needed.

was also used to prove lower bounds in Chapters 5 and 6. However, the particular covering strategy being used here is different from those applied in earlier papers and in Chapters 5 and 6, as it strongly exploits the fact that nonatomic writes may occur for arbitrary durations.

As explained above, at each induction step, we append one solo segment per each active process (that is not erased). Therefore, after $m-1$ induction steps (for some $m > 2$), a regular computation H can be decomposed into $m-1$ *segments* H^1, H^2, $\ldots$, H^{m-1}, where each H^j consists of the events that are appended at the j^{th} induction step (and are not erased so far).[7] Thus, the structure of H is as shown in Figure 7.4.

Here, $\text{Act}(H) = \{p_1,\ p_2,\ \ldots,\ p_k\}$ is the set of active processes. We also assume that H satisfies the following property.

- For each variable v, a regular computation H may contain at most one process that executes "uncovered" write(s) of v. (7.1)

Informally, an uncovered write to v (by a process p) is a write to which the covering strategy is not applied. Thus, if some other process q reads v later, then they may gain knowledge of p, which is clearly undesirable. Property (7.1) limits the number of uncovered writes, so that we can prevent such a case from happening without too much difficulty, as explained shortly. (This property is also formally stated in R4, given later.)

As explained above, most active processes have a "next" non-transition solo segment that can be potentially executed after H. For simplicity, assume that this is the case for all active processes. Thus, each active process p_j has an m^{th} solo segment $S(p_j, m)$. We now present examples that demonstrate why and when the two strategies — covering and erasing — are necessary.

[7]We consider the initial computation H_1 to have taken one induction step to construct. Thus, a computation after $m-1$ induction steps have $m-1$ solo segments per each active process.

(a)

$$\underbrace{\begin{array}{llll} S(p_1,1) & & & \\ & S(p_2,1) & & \\ & & \ddots & \\ & & & S(p_k,1) \end{array} \quad \cdots \quad \begin{array}{llll} S(p_1,m-1) & & & \\ & S(p_2,m-1) & & \\ & & \ddots & \\ & & & S(p_k,m-1) \end{array}}_{H = H^1 \circ H^2 \circ \cdots \circ H^{m-1}} \quad \underbrace{\begin{array}{llll} S(p_1,m) & & & \\ & S(p_2,m) & & \\ & & \ddots & \\ & & & S(p_k,m) \end{array}}_{\text{H^m: consists of newly appended solo segments}}$$

(b)

$$\underbrace{\begin{array}{llll} S(p_2,1) & & & \\ & S(p_4,1) & & \\ & & \ddots & \\ & & & S(p_k,1) \end{array} \quad \cdots \quad \begin{array}{llll} S(p_2,m-1) & & & \\ & S(p_4,m-1) & & \\ & & \ddots & \\ & & & S(p_k,m-1) \end{array}}_{\text{H, with odd-numbered active processes erased}} \quad \underbrace{\begin{array}{llll} S(p_2,m) & & & \\ & S(p_4,m) & & \\ & & \ddots & \\ & & & S(p_k,m) \end{array}}_{H^m}$$

(c)

$$\underbrace{\begin{array}{lll} S(p_1,1) & & \\ & \ddots & \\ & & S(p_k,1) \end{array} \quad \cdots \quad \begin{array}{lll} S(p_1,m-1) & & \\ & \ddots & \\ & & S(p_k,m-1) \end{array}}_{H} \quad \underbrace{\begin{array}{lll} S(p_1,m) & & \\ & \ddots & \\ & & S(p_k,m) \end{array}}_{\text{H^m: each segment writes v}} \quad \underbrace{\begin{array}{lll} S(p_1,m+1) & & \\ & \ddots & \\ & & S(p_k,m+1) \end{array}}_{\text{H^{m+1}: each segment reads and writes v}}$$

Figure 7.5: Extensions of H: only relevant details are shown. **(a)** An ideal case. Each next critical event $ce(p_j, m)$ accesses a distinct variable v_j, which is not accessed in H. **(b)** Erasing strategy. Each next critical event $ce(p_j, m)$ writes variable $v_{\lceil j/2 \rceil}$, which is not accessed in H. **(c)** A case in which the covering strategy is necessary. Each next critical event $ce(p_j, m)$ writes the same variable v, and each $(m+1)^{\text{st}}$ critical event $ce(p_j, m+1)$ reads v. Each $(m+1)^{\text{st}}$ solo segment $S(p_j, m+1)$ also contains noncritical write(s) of v. Note that this computation incurs information flow.

Example of the erasing strategy. First, we consider an "ideal" case, shown in Figure 7.5(a), in which each next critical event $ce(p_j, m)$ accesses a distinct variable v_j. Moreover, we assume that each v_j is *not* accessed in H. (As explained in detail later, this condition can be ensured by erasing some processes.) In this case, we simply append all of the next solo segments. Since each critical event accesses a distinct variable, they cannot induce information flow. (For now, we ignore the possibility that *noncritical* events in these next solo segments may induce information flow. We will address that issue later.) Thus, we can construct a longer regular computation with a (partial) structure given in Figure 7.5(a).

In this ideal case, no erasing or covering is necessary. However, consider another case, depicted in Figure 7.5(b), in which k active processes access $k/2$ distinct variables, where for simplicity, k is assumed to be even. We assume that each variable v_j (for $1 \leq j \leq k/2$) is written by both $ce(p_{2j-1}, m)$ and $ce(p_{2j}, m)$. In this case, we cannot append both $S(p_{2j-1}, m)$ and $S(p_{2j}, m)$, because then Property (7.1) would be violated. Thus, we apply the erasing strategy: we erase, say, every odd-numbered active process, and construct a regular computation with $k/2$ active processes, as shown in Figure 7.5(b).

If the next critical events collectively access many distinct variables, then we can apply this erasing strategy in the obvious way (by selecting one process for each variable and erasing the rest), and obtain a longer regular computation with enough active processes. Thus, for each variable v of concern, there exists exactly one process that accesses v, and hence information flow is precluded and (7.1) is preserved. (As explained later, we can also ensure that every write to v in H (if any) is properly covered, with some additional erasing.) However, if the next critical events collectively write only a small number of variables, this strategy may leave too few active processes, and the induction may stop before the desired lower bound is achieved. We now consider an example of such a situation.

Example of the covering strategy. As a stepping stone toward a general covering strategy, we present here a simplified version of the basic technique. If the majority of the next critical events are reads, then we may prevent information flow as follows: **(i)** we erase each process that has a write as its next critical event, and **(ii)** for each variable v that is read by some next critical read event, we erase a process (if any) that executes uncovered write(s) of v in H. (By (7.1), for each such v, we erase at most one process.) We thus ensure that each next critical read event reads the initial value of the variable it accesses.

On the other hand, if the majority of the next critical events are writes, then we apply the covering strategy. For simplicity, assume that every next critical event writes v. (That is, $ce(p_j, m)$ is a write event of v, for each $p_j \in \text{Act}(H)$.) In this case, appending all next solo segments as in Figure 7.5(a) may lead to information flow in further induction steps. To see why, suppose that each p_j in our example reads v in its $(m+1)^{\text{st}}$ critical event $ce(p_j, m+1)$. Moreover, suppose that each $(m+1)^{\text{st}}$ solo segment $S(p_j, m+1)$ contains noncritical write(s) of v. This situation is depicted in Figure 7.5(c).

In this case, every process, in its $(m+1)^{\text{st}}$ segment, reads from v a value written by another process. For example, process p_1 reads from v the value written by p_k in $S(p_k, m)$. Erasing p_k will not eliminate this information flow, because then p_1 will read the value written by p_{k-1} instead.

Information flow may be eliminated by changing some critical writes into invocation events on the same variable. Since the $(m+1)^{\text{st}}$ solo segments contain noncritical writes of v, we must include an invocation event on v after each $(m+1)^{\text{st}}$ solo segment (that is not erased) in order to cover such writes. (We again assume that k is even, for simplicity.) By stalling half of the active processes (say, the even-numbered ones) and letting the other half continue their active execution, we append the following computations at the m^{th} and $(m+1)^{\text{st}}$ steps, respectively:

$$\begin{aligned}
H^m &= S(p_1, m) \circ S(p_3, m) \circ \cdots \circ S(p_{k-1}, m) \circ \langle ie(p_k, m) \rangle, \\
H^{m+1} &= S(p_1, m+1) \circ \langle ie(p_2, m) \rangle \circ S(p_3, m+1) \circ \langle ie(p_4, m) \rangle \circ \cdots \circ \\
&\qquad S(p_{k-3}, m+1) \circ \langle ie(p_{k-2}, m) \rangle \circ S(p_{k-1}, m+1).
\end{aligned}$$

Here, H^m consists of the solo segments of all odd-numbered active processes, followed by an invocation event by p_k on v (so that the following critical read of v in $S(p_1, m+1)$ does not gain knowledge of p_{k-1}). In H^{m+1}, solo segments by odd-numbered active processes are interleaved with invocation events on v by even-numbered processes, so that information flow among them is prevented. We thus guarantee that any read from v by a process p_j (for odd j) either reads a value written by p_j, or happens concurrently with a nonatomic write to v (by some covering process). In the latter case, by our system model, any value may be read. Thus, information flow can be prevented by assuming that each such process p_j reads the same value as in its solo computation.

After appending both H^m and H^{m+1}, we thus have $k/2$ active processes (the odd-numbered ones), plus $k/2$ covering processes (the even-numbered ones) that have been used in covering v and do not participate in further induction steps.

Unfortunately, the construction above is somewhat simplified and does not really work. This is because, in further induction steps (beyond the $(m+1)^{\text{st}}$), we may append additional solo segments $S(p_j, l)$ (for odd j and $l > m+1$) that contain both noncritical reads and writes of v. Thus, they too must be interleaved with invocation events on v to prevent information flow, but we do not have any more "available" covering processes that may execute these events.

In order to solve this problem, we do not stall only half of the active processes, but "most" of them: for each active process (that is not stalled), we stall s processes (where $s = \Omega(\log N/ \log\log N)$), so that there are enough invocation events to insert in further induction steps. (We thus reduce the number of active processes by a factor of $s + 1$.) We then insert an invocation event after *every* solo segment $S(p_j, l)$ such that $l \geq m$. (Note that this may introduce a large number of unnecessary invocation events, as explained shortly.) Thus, segments appended at the m^{th} and later steps may have the following structure.

$$\begin{array}{rcl} H^m & = & S(p_1, m) \circ \langle ie(p_2, m)\rangle \circ S(p_{s+1}, m) \circ \langle ie(p_{s+2}, m)\rangle \circ \cdots \\ H^{m+1} & = & S(p_1, m+1) \circ \langle ie(p_3, m)\rangle \circ S(p_{s+1}, m+1) \circ \langle ie(p_{s+3}, m)\rangle \circ \cdots \\ H^{m+2} & = & S(p_1, m+2) \circ \langle ie(p_4, m)\rangle \circ S(p_{s+1}, m+2) \circ \langle ie(p_{s+4}, m)\rangle \circ \cdots \\ \cdots & & \cdots \end{array} \tag{7.2}$$

Note that some of these invocation events are actually unnecessary. For example, event $ie(p_2, m)$ is unnecessary because the following segment $S(p_{s+1}, m)$ starts with a write to v (*i.e.*, $ce(p_{s+1}, m)$), thus overwriting the value written by $S(p_1, m)$. Also, for any $l > m$, the solo segment $S(p_j, l)$ does not necessarily contain a write to v. Thus, it may be overkill to insert an invocation event after every $S(p_j, l)$. We simply include such unnecessary invocation events to simplify bookkeeping.

A generic description of the covering strategy. The structure depicted in (7.2) is still simplified, for three reasons. First, solo segments may contain writes to variables other than v, in which case invocation events on these variables will be placed together with invocation events on v. Second, in practice, the m^{th} critical events may collectively access multiple variables. In that case, we have to apply the covering strategy separately to each variable.

Finally, it may be impossible to index and arrange processes in a regular fashion as above, since some of these active processes may be erased later. Thus, we need a more dynamic approach, as depicted in Figure 7.6. Assume that, at the m^{th} induction step, we find that there are "too many" processes whose m^{th} critical event is a write to v, as shown in Figure 7.6(a). (In (7.2), these processes comprise all active processes; in general, they will form a subset of the active processes.) We partition these processes into two sets: AW_v^m, the set of "active writers," and CW_v^m, the set of "covering writers." Processes in AW_v^m continue active execution, while processes in CW_v^m are stalled just before they execute their critical writes of v, and may later execute invocation events on v (Figure 7.6(b); the "reserve writers" in this figure will be considered later). (For example, in (7.2), processes p_1 and p_{s+1} belong to AW_v^m, while processes $p_2,\ p_3,\ p_4,\ p_{s+2},\ p_{s+3}$, and p_{s+4} belong to CW_v^m.) We say that we *select* processes in CW_v^m *for covering* variable v.

For each process $p \in AW_v^m$, we append its m^{th} solo segment $S(p, m)$ to construct the m^{th} segment. Since $S(p, m)$ contains $ce(p, m)$, a write to v, we have to cover this write. Thus, we choose a process q from CW_v^m and append $ie(q, m)$, q's invocation event on v, after $S(p, m)$ (Figure 7.6(b)). We say that we *deploy* a process q from CW_v^m in order to cover $S(p, m)$.

We now consider the l^{th} induction step, where $l > m$ (Figure 7.6(c)). For each process $p \in AW_v^m$ that is active at that point, we append its solo computation $S(p, l)$ in order to construct the l^{th} segment. Since $ce(p, m)$ writes v, $S(p, l)$ may contain a write to v. Thus, we choose a process q' from CW_v^m and append $ie(q, m)$ after $S(p, l)$. As before, we say that we deploy q' in order to cover $S(p, l)$. (As explained before, $ie(q', m)$ may in fact be unnecessary.)

Thus, if yet another process r reads v later, then r cannot read the value written by p in $S(p, l)$. In particular, if r's read is concurrent with the nonatomic write by q', then by our system model, r may read any value. Otherwise, the nonatomic write by q' has been terminated by yet another (atomic or nonatomic) write of v. (Recall that explicit response events are not used in our proof.) Thus, the value written by p is already overwritten. By repeating this argument for each reader and covered writer of v in H, it follows that covered writes cannot cause information flow. (This argument is formalized in Lemma D.1 in Appendix D.)

Note that, after the construction of the each segment, many processes in CW_v^m are left unused — that is, they are not deployed yet. These processes constitute RW_v^m, the set of *reserve processes* (or "reserve writers"). (See Figure 7.6(b) and (c).) The processes in RW_v^m serve two purposes. First, when we inductively construct longer computation(s), these processes are deployed to cover v after newly appended solo segments. For example, process q' is a reserve process in H_m (the computation obtained at the end of the m^{th} step, depicted in Figure 7.6(b)) but not in H_l (depicted in Figure 7.6(c)). Second, if some deployed process in $(CW_v^m - RW_v^m)$ is erased later (due to a conflict via some other variable), then a process in RW_v^m is selected to take its place. For example, in Figure 7.7 (which is a continuation of Figure 7.6), process q' is erased, and we choose a process r from RW_v^m and use r to take the place of q' in covering v.

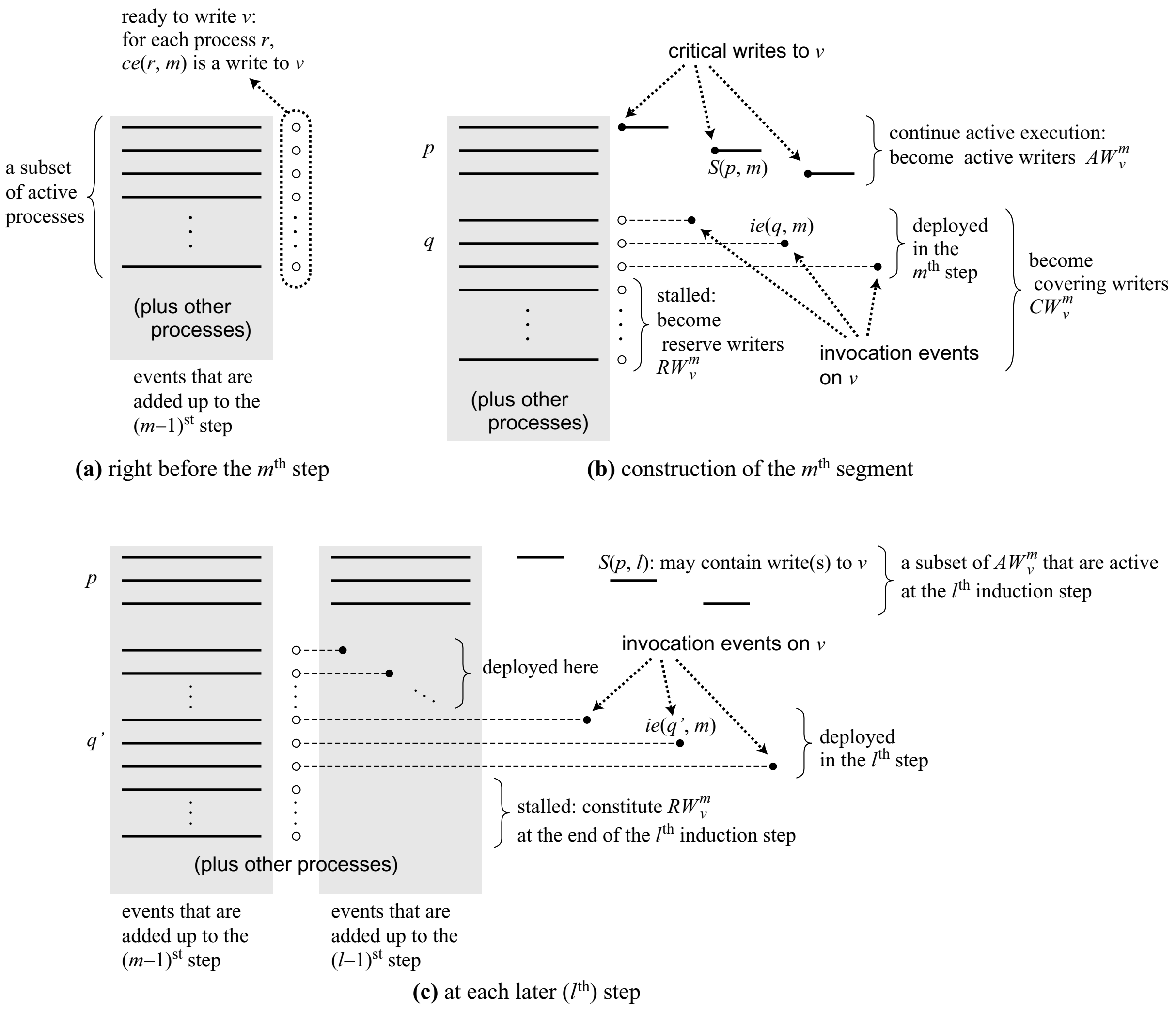

ready to write v:
for each process r, $ce(r, m)$ is a write to v
a subset of active processes
(plus other processes)
events that are added up to the $(m-1)^{st}$ step
(a) right before the m^{th} step
critical writes to v
p
q
$S(p, m)$
continue active execution: become active writers AW_v^m
$ie(q, m)$
deployed in the m^{th} step
become covering writers CW_v^m
stalled: become reserve writers RW_v^m
invocation events on v
(plus other processes)
(b) construction of the m^{th} segment
p
q'
$S(p, l)$: may contain write(s) to v
a subset of AW_v^m that are active at the l^{th} induction step
invocation events on v
deployed here
$ie(q', m)$
deployed in the l^{th} step
stalled: constitute RW_v^m at the end of the l^{th} induction step
(plus other processes)
events that are added up to the $(m-1)^{st}$ step
events that are added up to the $(l-1)^{st}$ step
(c) at each later (l^{th}) step

Figure 7.6: Covering strategy. We only show relevant processes. (In general, the computation has many other processes that are not depicted here.) Here and in later figures, horizontal lines depict events of a particular process, black circles (●) depict a single event, and empty circles (○) depict an event that is *enabled* at that point but not executed. **(a)** At the m^{th} step, we find "too many" active processes that are ready to write variable v. **(b)** At the same step, we stall some of these processes — these processes constitute the set of "covering writers" CW_v^m. The rest of the processes constitute the set of "active writers" AW_v^m — these processes remain active and continue execution. Some processes among CW_v^m are deployed to cover the m^{th} solo segments. For example, q is deployed to cover $S(p, m)$. The rest of CW_v^m remain undeployed and constitute RW_v^m. We thus construct H_m. **(c)** Construction of H_l at the l^{th} step (where $l > m$). In general, a subset of AW_v^m is active here. We also deploy a process q' from CW_v^m to cover $S(p, l)$.

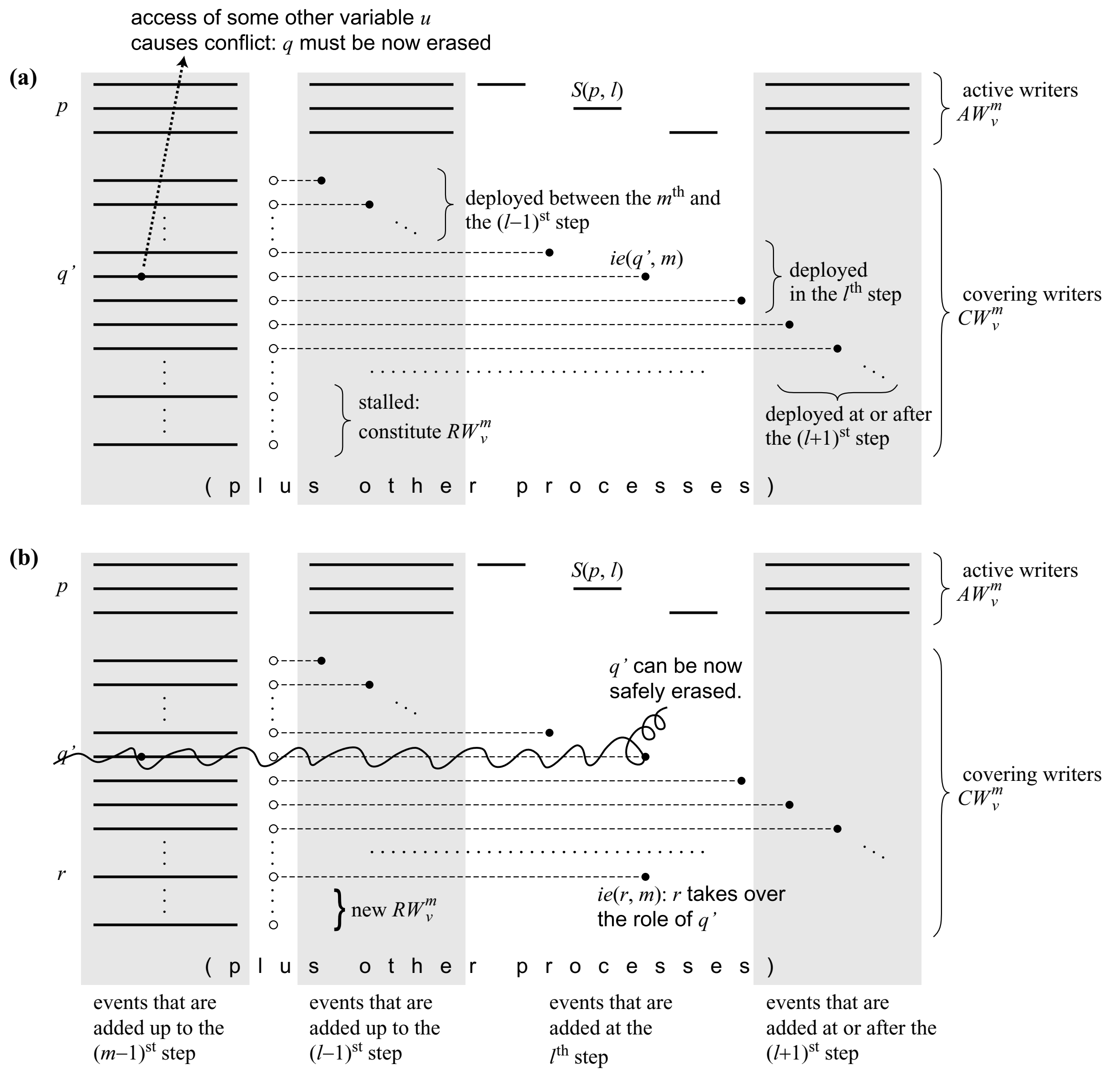

access of some other variable u causes conflict: q must be now erased
(a)
p
q'
$S(p, l)$
active writers AW_v^m
deployed between the mth and the $(l-1)$st step
$ie(q', m)$
deployed in the lth step
covering writers CW_v^m
stalled: constitute RW_v^m
deployed at or after the $(l+1)$st step
(plus other processes)
(b)
q' can be now safely erased.
r
new RW_v^m
$ie(r, m)$: r takes over the role of q'
events that are added up to the $(m-1)$st step
events that are added up to the $(l-1)$st step
events that are added at the lth step
events that are added at or after the $(l+1)$st step

Figure 7.7: The use of reserve processes to "exchange" two processes before erasing. **(a)** After Figure 7.6(c), at some later step, we find that process $q' \in CW_v^m$ incurs a conflict via some other variable u. Thus, we have to erase q'. **(b)** We choose some process r from RW_v^m, and let r execute its invocation event in place of q'. Process q' can now be safely erased. Note that $|RW_v^m|$ is reduced by one, since r is no longer a reserve process.

In practice, additional complications arise if a variable is chosen multiple times for covering throughout the induction. For example, we may find that many processes write v at the mth induction step, and partition them into two sets AW_v^m and CW_v^m, as described above. Later, at the kth induction step, we may again find that many processes (that have not written v so far) write v. Since they did not write v at the mth induction step, they are clearly disjoint from both AW_v^m and CW_v^m. We thus partition these processes and construct two sets AW_v^k and CW_v^k. In this case, each active process that writes v is covered by its corresponding subset of covering processes: if $p \in AW_v^m$ and $p' \in AW_v^k$ hold, then we cover $S(p, l)$ for each $l \geq m$ (respectively, $S(p', l')$ for each $l' \geq k$) by deploying some process from CW_v^m (respectively, CW_v^k).

7.3.2 Formal Definitions

Having outlined some of the basic ideas of our proof, we now define some relevant notation and terminology. At the core of these definitions is the notion of a regular computation, mentioned above. After formally defining the class of regular computations, we give a detailed proof sketch.

A regular computation H has an associated *induction number* m_H, which is the number of induction steps taken to construct H. Such a computation H can be written $H = H^1 \circ H^2 \circ \cdots \circ H^{m_H}$, where H^m is called the m^{th} *segment* of H. For each *segment index* m ($1 \leq m \leq m_H$), H^m consists of the events that are appended at the m^{th} induction step (and are not erased so far), and contains exactly one critical event by each process that was active at the m^{th} induction step. We now explain the process groups involved in constructing H in detail.

We define $P(H)$, the set of *participating processes* in H, as follows:

$$P(H) = \{p \in P\colon H \mid p \neq \langle\rangle\}. \tag{7.3}$$

Processes in $P(H)$ are partitioned into two sets: $\mathrm{Act}(H)$, the *active processes*, and $\mathrm{Cvr}(H)$, the *covering processes*.

$$P(H) = \mathrm{Act}(H) \cup \mathrm{Cvr}(H) \quad \wedge \quad \mathrm{Act}(H) \cap \mathrm{Cvr}(H) = \{\}. \tag{7.4}$$

As explained above, each covering process p is selected to cover some variable v at some induction step m, and is stalled right before executing its next critical event $ce(p, m)$, which must be a write to v. In this case, we define the *covering index* of p, denoted $ci(p)$, to be m, and the *covering variable* of p, denoted $cv(p)$, to be v. We also define the set CW_v^m as the set of covering processes (or "covering writers") that are selected at the m^{th} induction step to cover v:

$$CW_v^m = \{p \in \mathrm{Cvr}(H)\colon (ci(p), cv(p)) = (m, v)\}. \tag{7.5}$$

By definition, we also have the following:

$$(ci(p), cv(p)) = (m, v) \quad \text{only if} \quad ce(p, m) \text{ is a write to } v. \tag{7.6}$$

As explained before, a process q in CW_v^m may be deployed to cover $S(p, l)$, for some process p and segment index $l \geq m$. In this case, we define $cp(q)$, the process covered by q, to be p. We let $cp(q) = \perp$ if q is not deployed in H.

In our proof, no processes are selected for covering at the first induction step:

$$ci(p) \geq 2, \quad \text{for each } p \in \text{Cvr}(H). \tag{7.7}$$

The covering processes may also be grouped as follows: we define $\text{Cvr}^m(H)$, the set of covering processes *at the end of the m^{th} segment*, as follows:

$$\text{Cvr}^m(H) = \{p \in \text{Cvr}(H)\colon ci(p) \leq m\} = \bigcup_{1 \leq j \leq m,\ v \in V} CW_v^j. \tag{7.8}$$

$\text{Cvr}^m(H)$ consists of processes that are selected for covering through the m^{th} induction step. We similarly define $\text{Act}^m(H)$, the set of active processes *at the end of the m^{th} segment*, as follows:

$$\text{Act}^m(H) = P(H) - \text{Cvr}^m(H). \tag{7.9}$$

$\text{Act}^m(H)$ consists of processes that have *not* been selected for covering, and hence are active at the end of the m^{th} induction step. A process p in $\text{Act}^m(H)$ may be selected for covering in some later, say, l^{th}, induction step, in which case p belongs to both $\text{Act}^m(H)$ and $\text{Cvr}^l(H)$ (see Figure 7.8). Note that if a process q is selected to cover a variable v at the m^{th} induction step (*i.e.*, $q \in CW_v^m$), then q does not become active again. That is, $q \in \text{Cvr}^m(H)$ implies $q \notin \text{Act}^{m'}(H)$, for each $m' \geq m$.

From the description above, we have $\text{Cvr}(H) = \text{Cvr}^{m_H}(H)$ and $\text{Act}(H) = \text{Act}^{m_H}(H)$. Note that, by (7.7), (7.8), and (7.9), the following hold:

$$\text{Cvr}^1(H) = \{\} \quad \text{and} \quad \text{Act}^1(H) = P(H). \tag{7.10}$$

We now describe the structure of H^l, the l^{th} segment of H. We can write H^l as follows:

$$H^l = S(p_1, l) \circ C(p_1, l; H) \circ S(p_2, l) \circ C(p_2, l; H) \circ \cdots \circ S(p_k, l) \circ C(p_2, l; H), \\ \text{where } \{p_1, p_2, \ldots, p_k\} = \mathrm{Act}^l(H). \tag{7.11}$$

$S(p, l)$, the lth solo segment of p, was already defined in Section 7.2. We call $C(p, l; H)$ the lth *covering segment* of p. Computation $C(p, l; H)$ consists of invocation event(s) that cover writes contained in $S(p, l)$ (see (7.2) for a simple example). When there is no possibility of confusion, we use $C(p, l)$ as a shorthand for $C(p, l; H)$. We also define AW_v^l, the set of *active writers* of v at the lth step, as follows:

$$AW_v^l = \{p \in \mathrm{Act}^l(H)\colon \mathit{ce}(p, l) \text{ is a write to } v\}. \tag{7.12}$$

We now describe the structure of $C(p, l)$ in detail. For each $m \leq l$, $\mathit{ce}(p, m)$ may be a write to some variable v (*i.e.*, $p \in AW_v^m$ holds). If the covering strategy was applied at the mth induction step, then some processes have been chosen to cover v, that is, we have $CW_v^m \neq \{\}$. (See Figure 7.6(a).) Thus, as described before, we deploy a process q from CW_v^m, and let q execute its invocation event $\mathit{ie}(q, m)$ on v in $C(p, l)$. $C(p, m)$ consists of such invocation events, as stated formally in R1 and R2 below. (Thus, the invocation events depicted in Figure 7.6(b) and (c) are contained in covering segments, which have been omitted from the figure for simplicity. For example, $\mathit{ie}(q, m)$ and $\mathit{ie}(q', l)$ are contained in $C(p, m)$ and $C(p, l)$, respectively.)

We also define RW_v^m, the set of *reserve processes* (or "reserve writers"), to be the set of processes in CW_v^m that are not yet deployed:

$$\begin{aligned} RW_v^m &= \{q \in CW_v^m\colon q \text{ is not deployed in } H\} \\ &= \{q \in CW_v^m\colon \mathit{cp}(q) = \perp\}. \end{aligned} \tag{7.13}$$

When we consider multiple regular computations, we also write $CW_v^m(H)$, $AW_v^m(H)$, $RW_v^m(H)$, $\mathit{ci}(p; H)$, $\mathit{cv}(p; H)$, and $\mathit{cp}(p; H)$ in order to specify the relevant computation H.

The structure of a regular computation, explained so far, is depicted in Figure 7.8. We now formally define the notion of a regular computation. Conditions R1–R4 defined below are discussed after the definition.

Definition: Let $\mathcal{S} = (C, P, V)$ be a nonatomic one-shot mutual exclusion system. A computation H in C is *regular* if and only if it satisfies the following.

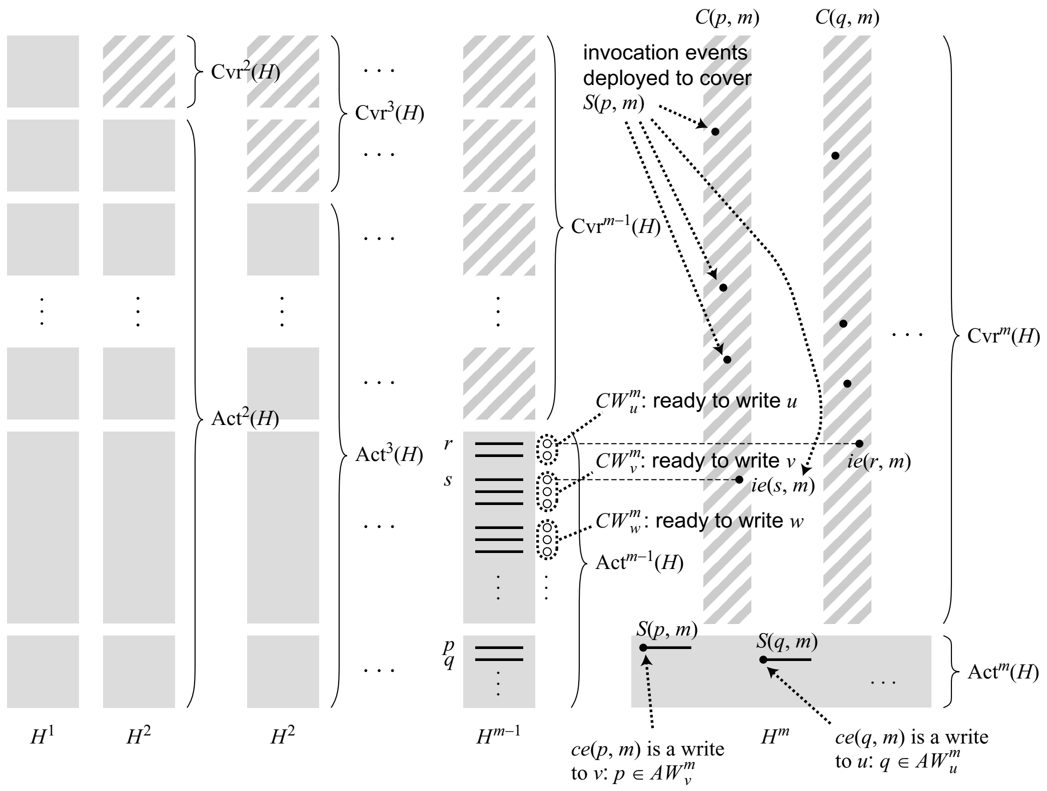


Figure 7.8: The structure of a regular computation. The computation is depicted as a collection of boxes, where the horizontal axis represents sequential order and the vertical axis represents different processes. A grey-filled box represents a collection of solo segment(s), and a striped box represents a collection of covering segment(s). For simplicity, segments $H^1, \ \ldots, \ H^{m-2}$ are shown simplified, and segments after H^m are not shown. The set of covering processes $(\mathrm{Cvr}^1(H), \ \mathrm{Cvr}^2(H), \ \ldots)$ increases, and the set of active processes decreases, as the segment index increases. We assume that the covering strategy is used at the m^{th} induction step. Thus, a number of processes (specifically, those in $\mathrm{Cvr}^m(H) - \mathrm{Cvr}^{m-1}(H)$) are stalled to cover the variables written by the m^{th} critical events of processes in $\mathrm{Act}^m(H)$. Processes in $\mathrm{Cvr}^m(H) - \mathrm{Cvr}^{m-1}(H)$ are partitioned into disjoints sets CW^m_u, CW^m_v, CW^m_w, *etc.*, and are ready to execute covering invocation events on the variables u, v, w, *etc.*, respectively. (To save space, subsets such as CW^m_v are depicted with only a few processes, but in reality these sets are much bigger.) A process $p \in \mathrm{Act}^m(H)$ executes its m^{th} solo segment $S(p, m)$. $S(p, m)$ is then followed by the covering segment $C(p, m)$, in which invocation events are executed to cover $S(p, m)$.

H can be written as $H = H^1 \circ H^2 \circ \cdots \circ H^{m_H}$, where H^m is called the m^{th} *segment* of H. Segment H^m consists of the events appended at the m^{th} induction step. We call m_H the *induction number* of H.

Processes in $P(H)$ are partitioned into $\mathrm{Cvr}(H)$ and $\mathrm{Act}(H)$. These sets, together with $CW_v^m(H)$, $\mathrm{Cvr}^m(H)$, $\mathrm{Act}^m(H)$, $AW_v^m(H)$, and $RW_v^m(H)$ are defined as in (7.3)–(7.5), (7.8), (7.9), (7.12), and (7.13). Segment H^m can be written as in (7.11). Moreover, H satisfies the following regularity conditions.

R1: For each event e_q contained in the *covering segment* $C(p, m)$, the following hold for some $j \leq m$ and variable v: $e_q = ie(q, j)$, $q \in CW_v^j$, $cp(q) = p$, and $ce(p, j)$ is a write to v (*i.e.*, $p \in AW_v^j$). (Note that $q \in CW_v^j$ implies that $ie(q, j)$ is an invocation event on v.)

R2: For each $p \in \mathrm{Act}^m(H)$ and $j \leq m$, if $ce(p, j)$ is a write to some variable v, and if CW_v^j is nonempty, then there is exactly one invocation event on v in $C(p, m)$, which must be $ie(q, j)$ for some $q \in CW_v^j$.

R3: H does not contain CS_p for any process p.

R4: Assume that, for some segment index m ($1 \leq m \leq m_H$) and process $p \in \mathrm{Act}^m(H)$, $ce(p, m)$ is a write to some variable v (*i.e.*, $p \in AW_v^m$) and CW_v^m is empty. (Note that, by (7.11), $p \in \mathrm{Act}^m(H)$ implies that H^m contains $S(p, m)$, and hence, $ce(p, m)$.) Then, for each segment index j and each process $q \in \mathrm{Act}^j(H)$ different from p, the following hold:

(i) if $j < m$ and $ce(q, j)$ is a write to v, then CW_v^j is nonempty (*i.e.*, q's write to v is covered);

(ii) if $j < m$ and $ce(q, j)$ is a read of v, then $q \in \mathrm{Cvr}^m(H)$ holds;

(iii) if $m \leq j \leq m_H$, then $ce(q, j)$ does not access v. □

Conditions R1 and R2 formally describe the structure of $C(p, m)$. Condition R3 is self-explanatory. We now explain Condition R4, which formalizes the requirement stated in (7.1). Informally, if a process p executes an "uncovered" critical write of v in segment H^m, then we require that p be the only uncovered writer of v throughout H. (Conditions (i) and (iii) of R4 imply that all other critical writes to v are covered.) In this case, we say that p is the *single writer* of v in H. The situation is depicted in Figure 7.9.

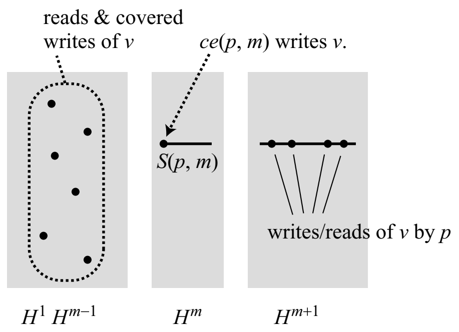


Figure 7.9: The single-writer case. We assume that $p \in \mathrm{Act}^m(H)$, $ce(p,m)$ is a write to v (*i.e.*, $p \in AW_v^m$), and that CW_v^m is empty. Segments H^1, H^2, ..., H^{m-1} may contain reads and covered writes of v. In segment H^m and later segments, p is the only process that may access v.

To see why this is necessary, assume that R4 is violated and H contains two uncovered writers of v, p and q. If yet another process r reads v in a future induction step, then erasing p does not eliminate information flow, because then r would read a value written by q instead. Thus, it becomes difficult to apply the erasing strategy without erasing too many processes. Condition R4 prevents such a case and simplifies bookkeeping.

We also require that no process other than p should read v after p writes v, because then such a process would gain knowledge of p. In particular, consider a process $q \neq p$ that reads v in some earlier segment H^k (where $k < m$). By the definition of a critical event, q must execute a critical read of v in or before H^k, *i.e.*, $ce(q,j)$ is a read of v for some $j \leq k < m$. In this case, Condition (ii) of R4 ensures $q \in \mathrm{Cvr}^m(H)$. To see why this is necessary, assume otherwise, *i.e.*, assume that $q \in \mathrm{Act}^m(H)$ holds. In that case, H^m would contain $S(q,m)$ (by (7.11)), which may in turn contain a *noncritical* read of v. Thus, q may gain knowledge of p by reading v, which is clearly unacceptable.

Finally, in order to simplify bookkeeping, we require that no process other than p should write v in or after H^m, as stated in Condition (iii) of R4.

We can thus define a "single writer" as follows.

Definition: Given a regular computation H and a variable v, we say that a process p is the *single writer* of v if $p \in AW_v^m(H)$ and $CW_v^m(H) = \{\}$ hold for some m. $\square$

It is easy to see that if a single writer exists, then it is uniquely defined. Assume, to the contrary, that we have two "single writers" p and q of v. Then, for some m and j, $p \in AW_v^m$, $q \in AW_v^j$, and $CW_v^m = CW_v^j = \{\}$ hold. Without loss of generality, assume

$m \leq j$. By (7.12), we have the following: $p \in \mathrm{Act}^m(H)$, $q \in \mathrm{Act}^j(H)$, $ce(p, m)$ writes v, and $ce(q, j)$ writes v. But this contradicts R4(iii).

Properties of a regular computation. We now define some additional properties of a regular computation that are used in our proof. In order to guarantee that the induction can continue, we need to ensure that there are enough covering processes for each covered variable *and* for each process writing that variable. Consider a process $p \in AW_v^m$. If p's write to v is covered (*i.e.*, CW_v^m is nonempty), then we need to deploy a process from CW_v^m to cover each of $S(p, m)$, $S(p, m+1)$, and so on. Since the induction continues until we construct $\Theta(\log N / \log\log N)$ segments, we may need up to $\Theta(\log N / \log\log N)$ covering processes in CW_v^m in order to cover p.[8] Therefore, we need $|AW_v^m| \cdot \Theta(\log N / \log\log N)$ processes in CW_v^m to ensure that we do not run out of covering processes during the induction.

To simplify bookkeeping, we assume some positive integer value $c = c(N)$, satisfying

$$c = \Theta(\log N), \tag{7.14}$$

and require that CW_v^m has at least $c \cdot |AW_v^m|$ processes. (The exact value of c is unimportant, since we are interested in an asymptotic lower bound. Throughout our proof, we assume the existence of a fixed one-shot mutual system, and hence we consider c as a fixed constant.)

However, this bound on $|CW_v^m|$ is still insufficient, since processes in CW_v^m may also be erased in induction steps beyond the m^{th}. (See Figure 7.7 for an example.) Thus, we actually need to ensure that $c \cdot |AW_v^m|$ covering processes *survive* even after some such processes are erased in future induction steps. As explained in detail later, we ensure that at most c processes are erased from CW_v^m at any induction step.[9] Thus, at the m^{th} step, we select $c \cdot (c - m)$ *additional* processes for covering, since we have at most $c - m$ induction steps beyond the m^{th}.

It follows that we need to select at least $c \cdot (|AW_v^m| + c - m)$ covering processes for CW_v^m, at the m^{th} induction step. Since H has taken total m_H steps to construct, $c \cdot (m_H - m)$ processes may have already been erased from CW_v^m (c processes for each step between $(m+1)^{\text{st}}$ and $m_H{}^{\text{th}}$). Therefore, we define $\mathrm{req}(m, v; H)$, the *required*

[8]If p is not active at the end of H (*i.e.*, $p \notin \mathrm{Act}(H)$), or if p becomes a covering process at some future induction step, then p requires fewer covering processes throughout the induction, since it has fewer solo segments in H or its extensions.

number of covering processes for $AW_v^m(H)$, as $c \cdot (|AW_v^m(H)| + c - m) - c \cdot (m_H - m)$:

$$\mathrm{req}(m, v; H) = c \cdot (|AW_v^m(H)| + c - m_H). \tag{7.15}$$

When there is no possibility of confusion, we use $\mathrm{req}(m, v)$ as a shorthand for $\mathrm{req}(m, v; H)$.

Given a regular computation H, for each pair (m, v), where $1 \le m \le m_H$ and $v \in V$, we define its *rank* $\pi(m, v; H)$ as follows:

$$\pi(m, v; H) = \begin{cases} \max\{0,\ \mathrm{req}(m, v) - |CW_v^m(H)|\}, & \\ \qquad \text{if } AW_v^m(H) \neq \{\} \ \wedge\ CW_v^m(H) \neq \{\}; & \\ 0, \qquad \text{otherwise.} & \end{cases} \tag{7.16}$$

When there is no possibility of confusion, we use $\pi(m, v)$ as a shorthand for $\pi(m, v; H)$. Note that $\pi(m, v)$ is always nonnegative.

We also say that a pair (m, v) of a segment index and a variable, where $1 \le m \le m_H$ and $v \in V$, is a *covering pair* if both AW_v^m and CW_v^m are nonempty. Thus, $\pi(m, v)$ is nonzero only if (m, v) is a covering pair. We also define the *maximum rank* $\pi_{\max}(H)$ and the *total rank* $\pi(H)$ of a regular computation H to be the maximum and the sum of its ranks:

$$\pi_{\max}(H) = \max_{1 \le m \le m_H,\ v \in V} \pi(m, v; H); \tag{7.17}$$

$$\begin{aligned} \pi(H) &= \sum_{1 \le m \le m_H,\ v \in V} \pi(m, v; H) \\ &= \sum_{(m,v):\ \text{covering pair in } H} \pi(m, v; H). \end{aligned} \tag{7.18}$$

Informally, a zero rank indicates that we have enough processes in CW_v^m to cover AW_v^m throughout the rest of the induction, while a positive rank indicates that CW_v^m is not large enough. We ensure that each induction step results in a regular computation H with a maximum rank of zero. (Thus, $\pi(m, v; H)$ is zero for all m and v.) Within a single the induction step, however, we may obtain intermediate computations with positive maximum ranks; if $\pi(m, v)$ becomes too high (for some m and v), then we erase some processes in AW_v^m to decrease $\mathrm{req}(m, v)$ and $\pi(m, v)$.

[9] In fact, more than c processes may be erased when it is safe to do so, *e.g.*, if CW_v^m is "large enough" to begin with. This will be explained in the detailed proof sketch.

7.3.3 Detailed Proof Overview

Initially, we start with a regular computation H_1 with induction number 1, in which $\mathrm{Act}(H_1) = P$, $\mathrm{Cvr}(H_1) = \{\}$, and each process $p \in P$ executes its first solo segment $\langle \mathit{Enter}_p \rangle$. At the $(m+1)^{\mathrm{st}}$ induction step, we consider a computation $H_m = H^1 \circ H^2 \circ \cdots \circ H^m$ such that $\mathrm{Act}(H_m)$ consists of n processes, each of which executes m critical events. As stated before, we also assume that H_m has a maximum rank of zero. By erasing some processes in H_m and appending a new $(m+1)^{\mathrm{st}}$ segment (as explained below), we can construct a regular computation H_{m+1} with a maximum rank of zero and induction number $m+1$, such that $\mathrm{Act}(H_{m+1})$ consists of $\Omega(n/c^3)$ $(= \Omega(n/\log^3 N))$ processes,[10] each of which executes $m+1$ solo segments.

By repeating the induction step, we construct a series of regular computations H_1, H_2, $\ldots$, H_k. The induction terminates when either $k = \Theta(c)$ is established or only one active process is left. In the former case, by (7.14), we have $k = \Theta(\log N)$. In the latter case, by combining the inequality $|\mathrm{Act}(H_{m+1})| = \Omega(|\mathrm{Act}(H_m)|/\log^3 N)$ over $m = 1, 2, \ldots, k-1$, and using $|\mathrm{Act}(H_k)| = 1$, we can show $k = \Omega(\log N/\log\log N)$. Therefore, in either case, we have $k = \Omega(\log N/\log\log N)$. Since each active process in H_k executes k solo segments (and hence, k critical events) in H_k, we have our lower bound.

We now explain the inductive construction method, which is formally described in Lemmas D.8 and D.9 in Appendix D.

Consider a regular computation $H = H_{m_H}$ with induction number m_H, in which n active processes participate. Moreover, assume

$$\pi_{\max}(H) = 0. \tag{7.19}$$

In the remainder of this section, we construct another regular computation $G = H_{m_H+1}$, with induction number $m_H + 1$ and maximum potential zero, in which $\Omega(n/c^3)$ active processes participate.

Every process in $\mathrm{Act}(H)$ has executed its first m_H solo segments, and hence is ready to execute its $(m_H+1)^{\mathrm{st}}$. For each $p \in \mathrm{Act}(H)$, we define its "next" critical event to be $ce(p, m_H+1)$, the event p is ready to execute after H.

[10] Recall that we use $\log n$ to denote $\log_2 n$ (base-2 logarithm), and use $\log^k n$ to denote $(\log_2 n)^k$ (see page 16).

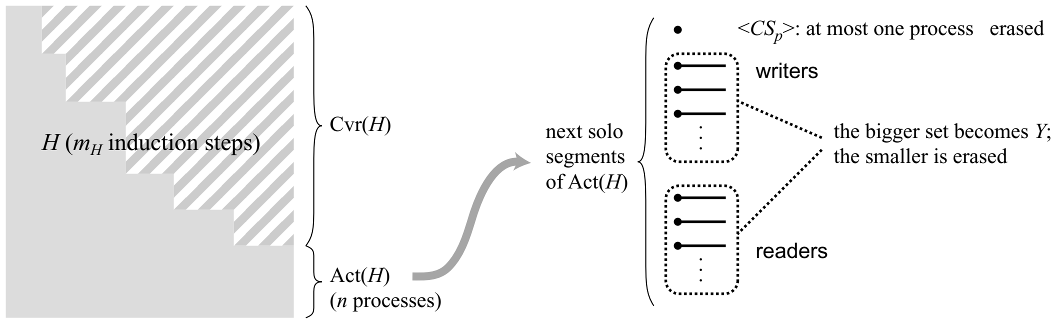

H (m_H induction steps)
Cvr(H)
Act(H)
(n processes)
next solo segments of Act(H)
<CS_p>: at most one process erased
writers
readers
the bigger set becomes Y; the smaller is erased

Figure 7.10: Construction of Y from Act(H). As in Figure 7.8, the grey-filled region represents a collection of solo segment(s), and the striped region represents a collection of covering segment(s). At most one process in Act(H) may execute its transition segment after H. We partition the rest of Act(H) into readers and writers, let Y be the larger of the two, and erase other active processes.

By the Exclusion property, it follows that at most one process p in Act(H) may execute a transition segment $\langle CS_p \rangle$. Thus, among the n processes in Act(H), at least $n - 1$ processes execute a non-transition solo segment after H.

These $n - 1$ processes can be partitioned into two subsets (one of which may be empty): the set of *readers*, which have a next critical event that is a read, and the set of *writers*, which have a next critical event that is a write. We define Y to be the larger of the two, and erase the smaller subset. (Thus, we have $|Y| \geq (n-1)/2 = \Theta(n)$; see Figure 7.10.) When we erase an active process p, we erase all of its solo segments $S(p, m)$ (for $1 \leq m \leq m_H$), as well as its covering segments $C(p, m)$. Thus, processes that are deployed to cover p's solo segments are turned into reserve processes.

The next critical events by processes in Y may generate *conflicts*, either with pre-existing events in H or among themselves. (A process or event conflicts with another if information flow is possible, or if a regularity condition is violated.) These conflicts must be eliminated in order to obtain the extended regular computation G.

First, consider conflicts *between* the next critical events and pre-existing events in H. In order to determine "who conflicts with whom," we define a "conflict mapping" K, defined over Y. For each $p \in Y$, we define $K(p) = q$ if p's next critical event $ce(p, m_H + 1)$ accesses variable v, and if q ($\neq p$) is the single writer of v in H (as depicted in Figure 7.9). (If H has no single writer of v, then we define $K(p) = \perp$.)

If $K(p) = q$, then we must erase either p or q, because appending p's next solo segment without erasing q would violate R4. Note that $P(H)$ may be partitioned into three subsets Cvr(H), Y, and Act(H) $- Y$. The set Act(H) $- Y$ has already been erased (see Figure 7.10). Thus, we need to consider only two kinds of conflicts:

conflicts *between* Y and $\mathrm{Cvr}(H)$, and conflicts *among* processes in Y. We now eliminate conflicts between Y and $\mathrm{Cvr}(H)$ by erasing each process in $CE = \{K(p)\colon p \in Y$ and $K(p) \in \mathrm{Cvr}(H)\}$, the "Covering processes to be Erased," as described below.

The chain erasing procedure. Consider a process $q \in CE$. Let $m = ci(q)$ and $v = cv(q)$. By (7.5), we have $q \in CW_v^m$. If q is not yet deployed in H (*i.e.*, $q \in RW_v^m$), then we can safely erase q without creating information flow. On the other hand, if q is already deployed to cover another process p, then we have to find some reserve process $r \in RW_v^m$, and "exchange" the role of q and r, before erasing q (see Figure 7.7). In this case, by (7.19), we can show $RW_v^m \neq \{\}$ (see Lemma D.5). Hence, we can always find an (undeployed) reserve process r. Thus, we can erase q. (This "exchange and erase" strategy is formally described in Lemma D.6.)

Although we can erase q and preserve regularity, erasing q reduces $|CW_v^m|$ by one and hence may increase $\pi(m, v)$. Informally, a high rank is problematic because there may not be enough reserve processes to continue further induction steps. Note that $\pi(m, v)$ satisfies the following properties, by (7.15) and (7.16).

- Erasing a process from CW_v^m increases $\pi(m, v)$ by at most one. (7.20)
- If we erase a process from AW_v^m while $\pi(m, v) = c$ holds, then $\pi(m, v) = 0$ is established. (7.21)

In order to bound the increase in $\pi(m, v)$, we apply the "chain erasing" procedure, as depicted in Figure 7.11; an example is illustrated in Figure 7.12. (This procedure is formally analyzed in Lemma D.7.) In this procedure, F denotes the computation being modified. Initially, F is the computation obtained by erasing $\mathrm{Act}(H) - Y$ from H (line 1 of Figure 7.11), and hence, by (7.19), we have $\pi_{\max}(F) = 0$. (Note that erasing an active process cannot increase any rank.)

During the chain erasing procedure, we want to maintain the invariant $\pi_{\max}(F) < c$. (As explained later, this allows us to construct an extended computation with a maximum rank of zero.) Assume that we erase a process $q \in CE \cap CW_v^m$ (line 4) while this invariant holds. If $\pi(m, v) < c - 1$ holds before erasing q, then, by (7.20), we can safely erase q while maintaining the invariant. Otherwise, $\pi(m, v) = c - 1$ holds before erasing q, and $\pi(m, v) = c$ is established. In this case, after erasing q, we select some process r from AW_v^m (line 6) and erase r (line 7). By (7.21), this will establish $\pi(m, v) = 0$, and thus the covering pair (m, v) now satisfies the invariant. However, if r is a covering process (at an induction step beyond the m^{th}), then this erasing may in

```
1: $F :=$ (the computation obtained by erasing $\mathrm{Act}(H) - Y$ from $H$);
2: for each process $p \in CE$ do
       /* loop invariant: $\pi_{\max}(F) < c$ */
3:     if $p \in P(F)$ then       /* is $p$ not yet erased? */
4:         erase $p$ from $F$; let the resulting computation be $F$;
5:         while $\pi_{\max}(F) = c$ do
               /* loop invariant: there exists exactly one covering pair $(m, v)$
                  satisfying $\pi(m, v; F) = c$ */
6:             choose a process $r$ from $AW_v^m(F)$;
7:             erase $r$ from $F$; let the resulting computation be $F$
   od fi od;
8: $H'' := F$
```

Figure 7.11: The chain-erasing procedure.

turn increase $\pi(ci(r), cv(r))$ by (7.20). If erasing r establishes $\pi(ci(r), cv(r)) = c$, then we have again established $\pi_{\max}(F) = c$. Thus, we have to erase yet another process from $AW_{cv(r)}^{ci(r)}$ (by executing lines 6 and 7 again). The chain erasing procedure continues in this manner as long as necessary.

Figure 7.12 shows two instances of such chain erasing. Note that processes r_1, r_4, and 59 belong to multiple subsets: for example, r_1 writes w in its second critical event, and is stalled before writing y at the third induction step. (Thus, we have $ci(r_1) = 3$ and $cv(r_1) = y$.) Processes 1..55 can be safely erased without violating the invariant. The chain erasing due to 56 is illustrated in insets (c) and (d); another chain erasing due to 57 is illustrated in insets (e)–(g).

Note that, in some cases, erasing a process $q \in CE$ results in erasing another process in $\mathrm{Act}(H)$, if we choose a process $r \in \mathrm{Act}(H)$ at line 6. If this "worst case" happens frequently, then the number of erased active processes may approach $|CE|$, and we may be left with too few active processes. This is clearly undesirable.

We claim that this worst case does not happen frequently. In fact, we can show that we erase at most $|CE|/(c-1)$ active processes. In order to show this, we first need the following two observations.

- Erasing a process q in CE may increase the total rank $\pi(F)$ by at most one. (7.22)
- Erasing a process r *not* in CE decreases $\pi(F)$ by at least $c - 1$. (7.23)

In order to prove (7.22), consider a covering pair (m, v). By definition, erasing q may change $\pi(m, v)$ only if either $q \in AW_v^m$ or $q \in CW_v^m$ holds. If $q \in AW_v^m$ holds, then by (7.15), $\mathrm{req}(m, v)$ is decreased by c, and hence $\pi(m, v)$ cannot increase. On the other hand, if $q \in CW_v^m$ holds, then by (7.16), $\pi(m, v)$ may increase by at most one.

(a)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\{1..29, 56, s_1..s_{70}\}$	$\{30..38, 57, s_{71}..s_{140}\}$	$\cdots$	$\{39..46, r_1, r_4, s_{141}..s_{200}\}$	$\cdots$	$\{47..55, 58, 59, s_{201}..s_{279}\}$	$\cdots$
$\|CW_v^m\|$	100	80	$\cdots$	70	$\cdots$	90	$\cdots$
AW_v^m	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	80	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(m, v)$	0	0	$\cdots$	0	$\cdots$	0	$\cdots$

(b)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\boldsymbol{\{56, s_1..s_{70}\}}$ (1..29 erased)	$\boldsymbol{\{57, s_{71}..s_{140}\}}$ (30..38 erased)	$\cdots$	$\boldsymbol{\{r_1, r_4, s_{141}..s_{200}\}}$ (39..46 erased)	$\cdots$	$\boldsymbol{\{58, 59, s_{201}..s_{279}\}}$ (47..55 erased)	$\cdots$
$\|CW_v^m\|$	**71**	**71**	$\cdots$	**62**	$\cdots$	**81**	$\cdots$
AW_v^m	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	80	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(m, v)$	**9**	**9**	$\cdots$	**8**	$\cdots$	**9**	$\cdots$

(c)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\boldsymbol{\{s_1..s_{70}\}}$ (56 erased)	$\{57, s_{71}..s_{141}\}$	$\cdots$	$\{r_1, r_4, s_{141}..s_{200}\}$	$\cdots$	$\{58, 59, s_{201}..s_{279}\}$	$\cdots$
$\|CW_v^m\|$	**70**	71	$\cdots$	62	$\cdots$	81	$\cdots$
AW_v^m	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	80	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(m, v)$	**10**	9	$\cdots$	8	$\cdots$	9	$\cdots$

(d)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\{s_1..s_{70}\}$	$\{57, s_{71}..s_{140}\}$	$\cdots$	$\boldsymbol{\{r_4, s_{141}..s_{200}\}}$ (r_1 erased)	$\cdots$	$\{58, 59, s_{201}..s_{279}\}$	$\cdots$
$\|CW_v^m\|$	70	71	$\cdots$	**61**	$\cdots$	81	$\cdots$
AW_v^m	$\boldsymbol{\{r_2, r_3\}}$ (r_1 erased)	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	**70**	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(m, v)$	**0**	9	$\cdots$	**9**	$\cdots$	9	$\cdots$

Figure 7.12: An example of chain erasing. In this figure, we assume $c = 10$ and $m_H = 5$. Thus, we also have req$(m, v) = 10 \cdot |AW_v^m| + 50$. We only show four covering pairs. Processes to be erased (*i.e.*, processes in CE) are denoted by sans serif numbers (1..59), and each other process is denoted as r_j (if it belongs to some set of active writers depicted here) or s_j (otherwise). Changes are marked with **boldface**. We assume that processes 1..59 are selected at line 2 sequentially. **(a)** Initial configuration before chain erasing starts. By (7.19), $\pi(m, v) = 0$ holds for all covering pairs. **(b)** After erasing processes 1..55. No chain erasing is necessary so far. **(c)** After erasing 56 (line 4 in Figure 7.11), we have $\pi(2, w) = 10 = c$. Thus, we find $\pi_{\max}(F) = c$ at line 5. Some process must be erased from AW_w^2. **(d)** We choose process r_1 at line 6, and erase it to reduce $\pi(2, w)$ to zero (line 7). This in turn increases $\pi(3, y)$ to 9, but we still maintain $\pi(3, y) < c$. Thus, we find $\pi_{\max}(F) < c$ at line 5, so no more chain erasing is necessary. (Continued on the next page.)

(e)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\{s_1..s_{70}\}$	$\boldsymbol{\{s_{71}..s_{140}\}}$ (57 erased)	$\cdots$	$\{r_4, s_{141}..s_{200}\}$	$\cdots$	$\{58, 59, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^m \rvert$	70	**70**	$\cdots$	61	$\cdots$	81	$\cdots$
AW_v^m	$\{r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	70	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(m, v)$	0	**10**	$\cdots$	9	$\cdots$	9	$\cdots$

(f)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\{s_1..s_{70}\}$	$\{s_{71}..s_{140}\}$	$\cdots$	$\boldsymbol{\{s_{141}..s_{200}\}}$ (r_4 erased)	$\cdots$	$\boldsymbol{\{58, s_{201}..s_{279}\}}$ (59 erased)	$\cdots$
$\lvert CW_v^m \rvert$	70	70	$\cdots$	**60**	$\cdots$	**80**	$\cdots$
AW_v^m	$\{r_2, r_3\}$	$\boldsymbol{\{r_5, r_6\}}$ (r_4 erased)	$\cdots$	$\boldsymbol{\{r_7\}}$ (59 erased)	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	70	**70**	$\cdots$	**60**	$\cdots$	90	$\cdots$
$\pi(m, v)$	0	**0**	$\cdots$	**0**	$\cdots$	**10**	$\cdots$

(g)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\{s_1..s_{70}\}$	$\{s_{71}..s_{140}\}$	$\cdots$	$\{s_{141}..s_{200}\}$	$\cdots$	$\{58, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^m \rvert$	70	70	$\cdots$	60	$\cdots$	80	$\cdots$
AW_v^m	$\{r_2, r_3\}$	$\{r_5, r_6\}$	$\cdots$	$\{r_7\}$	$\cdots$	$\boldsymbol{\{r_9, r_{10}, r_{11}\}}$ (r_8 erased)	$\cdots$
req(m, v)	70	70	$\cdots$	60	$\cdots$	**80**	$\cdots$
$\pi(m, v)$	0	0	$\cdots$	0	$\cdots$	**0**	$\cdots$

(h)

covering pair (m, v)	$(2, w)$	$(2, x)$	$\cdots$	$(3, y)$	$\cdots$	$(5, z)$	$\cdots$
CW_v^m	$\{s_1..s_{70}\}$	$\{s_{71}..s_{140}\}$	$\cdots$	$\{s_{141}..s_{200}\}$	$\cdots$	$\boldsymbol{\{s_{201}..s_{279}\}}$ (58 erased)	$\cdots$
$\lvert CW_v^m \rvert$	70	70	$\cdots$	60	$\cdots$	**79**	$\cdots$
AW_v^m	$\{r_2, r_3\}$	$\{r_5, r_6\}$	$\cdots$	$\{r_7\}$	$\cdots$	$\{r_9, r_{10}, r_{11}\}$	$\cdots$
req(m, v)	70	70	$\cdots$	60	$\cdots$	80	$\cdots$
$\pi(m, v)$	0	0	$\cdots$	0	$\cdots$	**1**	$\cdots$

Figure 7.12: An example of chain erasing, continued. **(e)** After erasing 57 (line 4), we have $\pi(2, x) = c$, and hence some process must be erased from AW_x^2. **(f)** We erase r_4 to reduce $\pi(2, x)$ to zero (line 7), which in turn establishes $\pi(3, y) = c$. Hence, we execute lines 6 and 7 again, and erase 59 to lower $\pi(3, y)$. (This is an example of a process in *CE* being erased at line 7.) However, this in turn establishes $\pi(5, z) = c$. **(g)** We erase r_8 to reduce $\pi(5, z)$ to zero. If r_8 is an active process, then we have established $\pi(F) < c$, and the inner loop (lines 5–7) terminates. Otherwise, $\pi(ci(r_8), cv(r_8))$ has been incremented by one; if its value reaches c, then the inner loop continues. **(h)** Finally, we assign $p :=$ 58 at line 2 and erase 58. No further chain erasing is necessary.

Since $q \in CW_v^m$ holds for at most one covering pair (m, v) (namely, $(ci(q), cv(q))$), it follows that erasing q may increase $\pi(F)$ by at most one.

We now prove (7.23). Note that a process $r \notin CE$ may be erased only at line 7. In this case, we have $r \in AW_v^m$ and $\pi(m, v) = c$ for some m and v, and by (7.22), $\pi(m, v)$ is reduced to zero. (For example, in Figure 7.12(d), erasing r_1 from $AW(2, w)$ reduces $\pi(2, w)$ from $c = 10$ to 0.) Thus, by erasing r, the rank of each covering pair (j, w) is changed as follows:

(i) if $(j, w) = (m, v)$, then $\pi(j, w)$ decreases by c;

(ii) otherwise, if $r \in AW^j_w$, then by (7.15) and (7.16), $\pi(j, w)$ cannot increase;

(iii) otherwise, if $r \in CW^j_w$, then by (7.16), $\pi(j, w)$ increases by at most one;

(iv) otherwise, $\pi(j, w)$ does not change.

Since $r \in CW^j_w$ holds for at most one covering pair (j, w), Case (iii) may apply to at most one covering pair. Note that, by (7.18), $\pi(F)$ is the sum of the ranks of all covering pairs. Therefore, by summing over Cases (i)–(iv), we have (7.23).

By (7.22), it follows that $\pi(F)$, being initially zero, increases by at most $|CE|$ throughout the execution of the chain erasing procedure. Since $\pi(F)$ is always nonnegative by definition, by (7.23), processes *not* in CE may be erased at most $|CE|/(c-1)$ times. Since $CE \subseteq \text{Cvr}(H)$ by definition, it follows that we erase at most $|CE|/(c-1)$ active processes in total.

Elimination of conflicts among active processes. Let H'' be the computation that results when the chain erasing procedure finishes, and let Y' be the subset of Y that was not erased during this procedure. As shown above, we have $|Y'| \geq |Y| - |Y|/(c-1) = \Theta(n)$. By erasing CE, we have eliminated conflicts *between* Y' and the covering processes. Moreover, the invariant of the chain erasing procedure ensures that $\pi_{\max}(H'') < c$ holds. By definition (given in (7.15)), for each covering pair (m, v), $\text{req}(m, v)$ is reduced by c when we append the new $(m_H + 1)^{\text{st}}$ segment. Hence, the rank $\pi(m, v)$, being less than c in H'', is reduced to zero after appending the new segment. It follows that all covering pairs in H'' will have zero rank in the extended computation G.

The remaining conflicts are divided into two categories: **(A)** conflicts *between* the next critical events (by processes in Y') and pre-existing events by (active) processes in Y', and **(B)** conflicts *among* the next critical events. We now eliminate conflicts of type A by constructing a conflict graph, in which each vertex is a process in Y' and each edge represents a conflict between two processes. That is, for each pair of processes p and q in Y', we introduce edge $\{p, q\}$ if and only if $K(p) = q \ \vee \ K(q) = p$ holds. (The definition of $K(p)$ was given on page 199.) Clearly, we introduce at most $|Y'|$ edges in total. The construction of $\mathcal{G}$ is shown in Figure 7.13.

We now want to find an independent set Z of $\mathcal{G}$, *i.e.*, a subset of the vertices such that no edge in $\mathcal{G}$ is incident to two vertices in Z. It is clear that such a set is free

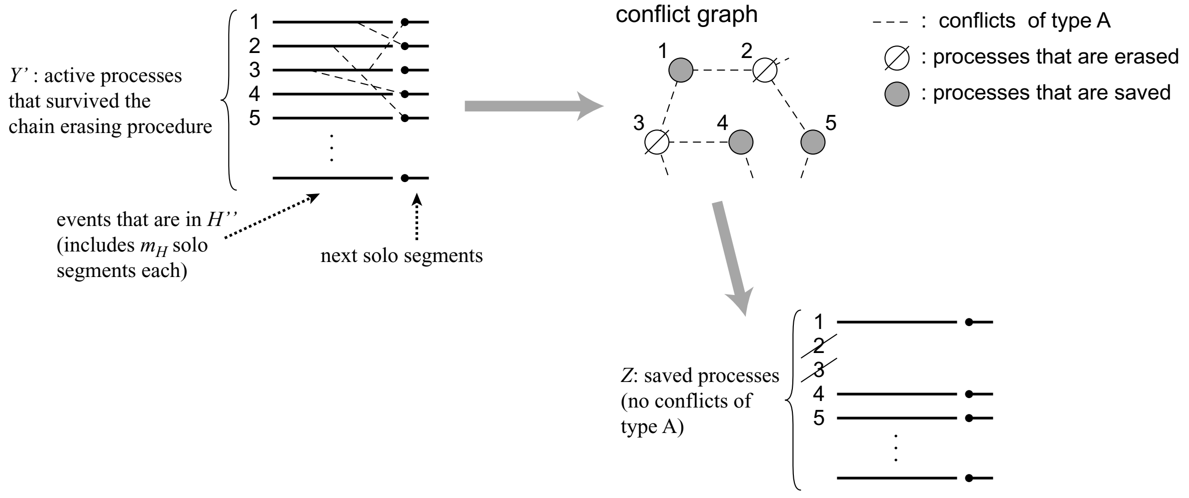

Y' : active processes that survived the chain erasing procedure
1
2
3
4
5
events that are in H''
(includes m_H solo segments each)
next solo segments
conflict graph
--- : conflicts of type A
: processes that are erased
: processes that are saved
Z: saved processes (no conflicts of type A)

Figure 7.13: Construction of the "conflict graph" $\mathcal{G}$. For simplicity, covering processes are not shown in this figure. In this figure, we assume $K(1) = 3$, $K(2) = 1$, $K(3) \notin Y'$, *etc.*

of conflicts of type A. Since $\mathcal{G}$ has $|Y'|$ vertices and at most $|Y'|$ edges, by applying Turán's Theorem (Theorem 5.1), we can obtain an independent set $Z \subseteq Y'$ with size $\Theta(|Y'|)$ $(= \Theta(n))$.

The processes in Z collectively execute $|Z| = \Theta(n)$ next critical events. Among the variables that are accessed by these events, we identify V_{HC}, the set of "high contention" variables, that are accessed by at least $4c^2$ next critical events. Similarly, we define V_{LC}, the set of "low contention" variables, as those that are accessed by at least one but less than $4c^2$ next critical events. (The constant factor 4 is needed in the covering strategy, described shortly.) Next, we partition the processes in Z, depending on whether their next critical events access a variable in V_{HC} or V_{LC}, as follows: $P_{\text{HC}} = \{p \in Z$: p's next critical event accesses some variable in $V_{\text{HC}}\}$, and $P_{\text{LC}} = Z - P_{\text{HC}}$.

Because $Z \subseteq \text{Act}(H)$, we can erase any process in Z and preserve regularity. We now have to eliminate conflicts of type B (by erasing some processes in Z), and determine which processes remain active and which processes are selected for covering, in order to construct the new $(m_H + 1)^{\text{st}}$ segment. Formally, we define two disjoint subsets Z^{Act} and Z^{Cvr} of Z: processes in Z^{Act} become active processes in the extended computation G, and processes in Z^{Cvr} become new covering processes to cover variables written by processes in Z^{Act}. (That is, we establish $\text{Act}(G) = Z^{\text{Act}}$ and $\text{Cvr}(G) = \text{Cvr}(H'') \cup Z^{\text{Cvr}}$.) Processes in $Z - (Z^{\text{Act}} \cup Z^{\text{Cvr}})$ (if any) are simply erased. The construction method is formally stated in Claim 2 in the proof of Lemma D.8, and is described below. (For each process $p \in Z^{\text{Act}}$, we also have to construct its covering segment $C(p, m_H + 1)$ by deploying appropriate processes. The detailed procedure for doing this, formally presented in Lemma D.9, is rather mechanical and is omitted here.)

Recall that Y consists of either all "writers" or all "readers" (see Figure 7.10). Since $Z \subseteq Y$, the same is true for Z. We now consider three cases.

Readers only. If a variable v is read by some process p in Z, then H (the original computation) satisfies one of the following three cases: **(i)** v is not written in H, **(ii)** all writes to v in H are covered, or **(iii)** v has a single writer q in H. In the first and and the second cases, the same conditions hold for H'', and hence p's read of v does not cause information flow. In the third case, we have $K(p) = q$, so q is already erased. In particular, if $q \in \text{Cvr}(H)$, then we have $q \in CE$ by definition, and hence q has been erased. On the other hand, assume that $q \in \text{Act}(H)$. Either q is erased in the chain erasing procedure, or $q \in Y$ holds. However, in the latter case, $\{p, q\} = \{p, K(p)\}$ is an edge in $\mathcal{G}$, and hence $p \in Z$ implies $q \notin Z$.

Therefore, by simply letting $Z^{\text{Act}} = Z$ and $Z^{\text{Cvr}} = \{\}$, we can construct G. Each next critical event by a process in Z reads the initial value of the variable it reads.

Erasing strategy. Assume that Z consists only of "writers," and that P_{LC} is larger than P_{HC}. In this case, we can erase P_{HC} and still have $\Theta(n)$ remaining active processes. Define V_{LC} as the set of variables accessed by the next critical events of P_{LC}. Then every variable in V_{LC} is a "low contention" variable, and hence is accessed by at most $4c^2$ different next critical events. It follows that V_{LC} contains at least $|P_{\text{LC}}|/4c^2 = \Theta(n/c^2)$ variables. By selecting one process for each such variable, we can create a set Z' of active processes, such that $|Z'| = \Theta(n/c^2)$, in which each next critical event accesses a distinct variable.

For each $p \in Z'$, we denote by $v_{\text{ce}}(p)$ the variable written by its next critical event $ce(p, m_H + 1)$. We want each process $p \in Z'$ to become the single writer $v_{\text{ce}}(p)$. Note that H'' does not contain a single writer of $v_{\text{ce}}(p)$, as shown above. In order to satisfy R4, we still must ensure that no active process in H'' reads $v_{\text{ce}}(p)$. Toward this goal, we create another conflict graph, as shown in Figure 7.14.

Since each process p in Z' executes m_H critical events in H, q may read at most m_H different variables. For each variable v read by p, we introduce edge $\{p, q\}$ if $q = v_{\text{ce}}(p)$. Since each $v_{\text{ce}}(p)$ is distinct (by the construction of Z'), we introduce at most m_H edges per each process in Z'. By applying Theorem 5.1 again, we can construct a subset Z^{Act}

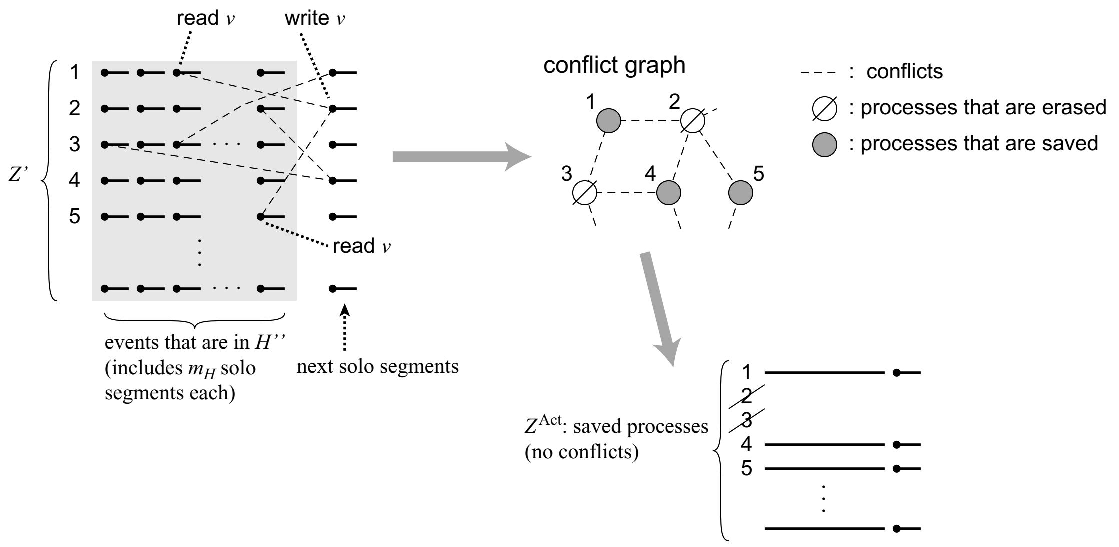

read v
write v
Z'
1
2
3
4
5
read v
events that are in H''
(includes m_H solo segments each)
next solo segments
conflict graph
- - - : conflicts
: processes that are erased
: processes that are saved
Z^{Act}: saved processes
(no conflicts)

Figure 7.14: Erasing strategy. For simplicity, covering processes are not shown in this figure. In this figure, we assume that the next critical event of process 2 writes v (*i.e.*, $v_{\text{ce}}(2) = v$), and that processes 1 and 5 read v in H''.

of Z' without any conflicts, such that $|Z^{\text{Act}}| = \Omega(|Z'|)/m_H = \Omega(n/c^3)$. (Here we use $m_H = O(c)$; otherwise, our lower bound is already attained.) We then define $Z^{\text{Cvr}} = \{\}$. Every process $p \in Z^{\text{Act}}$ becomes the single writer of $v_{\text{ce}}(p)$ in G.

Covering strategy. Assume that Z consists only of "writers," and that P_{HC} is larger than P_{LC}. In this case, we can erase P_{LC} and still have $\Theta(n)$ remaining active processes. Every next critical event by a process in P_{HC} writes a variable in V_{HC}. We now apply the covering strategy to each variable in V_{HC} (see Figure 7.6). Since each variable $v \in V_{\text{HC}}$ is written by at least $4c^2$ processes in P_{HC}, we can choose active writers $AW_v^{m_H+1}$ and covering writers $CW_v^{m_H+1}$ satisfying the following: **(i)** the number of all active writers is $\Omega(|Z|/c^2) = \Omega(n/c^2)$, and **(ii)** for each variable $v \in V_{\text{HC}}$, the rank $\pi(m_H + 1, v)$ is zero.

Formally, for each $v \in V_{\text{HC}}$, assume that $k(v)$ processes in P_{HC} write v. By assumption, we have $k(v) \geq 4c^2$. Among these processes, $a(v) = \lfloor k(v)/c^2 \rfloor - 1$ processes become active writers, and the rest become covering writers. From these two equations, we have the following inequalities, from which Conditions (i) and (ii) follow.

$$a(v) > \frac{k(v)}{c^2} - 2 \geq \frac{k(v)}{c^2} - \frac{k(v)}{2c^2} = \frac{k(v)}{2c^2};$$

$$
\begin{aligned}
|CW_v^{m_H+1}| &= k(v) - a(v) \\
&\geq c^2 \cdot (a(v)+1) - a(v) \\
&= (c^2-1) \cdot a(v) + c^2 \\
&\geq c \cdot a(v) + c \cdot (c - m_H) \\
&= \mathrm{req}(m_H+1, v).
\end{aligned}
$$

The collection of all active writers becomes Z^{Act}, and the collection of all covering writers becomes Z^{Cvr}. (Hence, we have $Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}} = P_{\mathrm{HC}}$.) Thus, we can construct G. □

The argument just explained (and proved formally in Appendix D) establishes the following theorem.

Theorem 7.2 *For any one-shot mutual exclusion system $\mathcal{S} = (C,\ P,\ V)$, there exist a p-computation F such that F does not contain CS_p, and p executes $\Omega(\log N/\log\log N)$ critical events in F, where $N = |P|$.* □

7.4 Concluding Remarks

We have presented a nonatomic local-spin mutual exclusion algorithm with $\Theta(\log N)$ worst-case RMR time complexity, which matches that of the best atomic algorithm (Algorithm YA-N) proposed to date. We have also shown that for any N-process nonatomic algorithm, there exists a single-process execution in which the lone competing process accesses $\Omega(\log N/\log\log N)$ distinct variables in order to enter its critical section. These bounds show that fast and adaptive algorithms are impossible if variable accesses are nonatomic, even if caching techniques are used to avoid accessing the processors-to-memory interconnection network.

CHAPTER 8

Generic Algorithm for *fetch-and-ϕ* Primitives*

In this chapter, we present a generic *fetch-and-ϕ*-based local-spin mutual exclusion algorithm with $O(1)$ RMR (remote-memory-reference) time complexity. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$, where N is the number of processes. As defined later, the *rank* of a *fetch-and-ϕ* primitive expresses the extent to which processes may "order themselves" using that primitive. By using an arbitration tree, a $\Theta(\log_r N)$ algorithm can be constructed using any primitive of rank r, where $2 \leq r < N$. For primitives that meet a certain additional condition, we present a $\Theta(\log N/\log\log N)$ algorithm, which is time-optimal for certain primitives of constant rank.

A *fetch-and-ϕ* primitive is characterized by a particular function ϕ (which we assume to be deterministic), and atomically accesses a single variable *var* as follows.

```
fetch-and-$\phi$($var$, $input$)
    $old := var$;
    $var := \phi(old, input)$;
    **return**($old$)
```

In this chapter, we distinguish between *fetch-and-ϕ* primitives that are comparison primitives and those that are not. As defined in Section 5.1, a *comparison primitive* conditionally updates a shared variable only if its value meets some condition; examples

*The results presented in this chapter have been published in the following paper.
[19] J. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and-ϕ primitives. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 538–547. IEEE, May 2003.

include *compare-and-swap* and *test-and-set*, defined in Section 2.1. Noncomparison primitives update variables unconditionally; examples include *fetch-and-increment* and *fetch-and-store*.

In Chapter 5, we established a time-complexity lower bound of $\Omega(\log N/\log\log N)$ RMRs for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives. In contrast, as discussed in Section 2.2.1, several $O(1)$ algorithms are known that are based on noncomparison *fetch-and-ϕ* primitives [23, 32, 38, 59, 61]. This suggests that noncomparison primitives may be the best choice to provide in hardware, if one is interested in implementing efficient blocking synchronization mechanisms.

From our earlier discussion of these $O(1)$ algorithms, recall the following.

- The algorithms of T. Anderson [23] and Graunke and Thakkar [38] use *fetch-and-increment* and *fetch-and-store*, respectively, and have $O(1)$ RMR time complexity only on CC machines.
- Mellor-Crummey and Scott's algorithm [61] has $O(1)$ RMR time complexity on both CC and DSM machines, but uses both *fetch-and-store* and *compare-and-swap*.
- Craig [32] and Landin and Hagersten [59] independently proposed the same algorithm, which is based on *fetch-and-store*. While Landin and Hagersten considered only CC machines, Craig presented constant-time variants of the algorithm for both CC and DSM machines.
- In other related work, Huang [42] proposed an algorithm based on *fetch-and-store* that has $O(1)$ *amortized* RMR time complexity on DSM machines.

Base on the existence of these algorithms, a number of intriguing questions were rasied in Section 1.4.4.

- Is it possible to devise an $O(1)$ algorithm for DSM machines that uses a single *fetch-and-ϕ* primitive other than *fetch-and-store*?
- Is it possible to automatically transform a local-spin algorithm for CC machines so that it has the same RMR time complexity on DSM machines?
- Given that the $\Omega(\log N/\log\log N)$ lower bound presented in Chapter 5 applies to algorithms that use comparison primitives, we know that there exist *fetch-and-ϕ*

primitives that are not sufficient for constructing $O(1)$ algorithms. For such primitives, what is the most efficient algorithm that can be devised?

- Can we devise a *ranking* of synchronization primitives that indicates the singular characteristic of a primitive that enables a certain RMR time complexity (for mutual exclusion) to be achieved?

Contributions. In this chapter, these questions are partially addressed. Our main contribution is a generic N-process *fetch-and-ϕ*-based local-spin mutual exclusion algorithm that has $O(1)$ RMR time complexity on both CC and DSM machines. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$. Informally, a primitive of rank r has sufficient symmetry-breaking power to linearly order up to r invocations of that primitive. Our generic algorithm breaks new ground because it shows that $O(1)$ RMR time complexity is possible using a wide range of primitives, on both CC and DSM machines. Thus, introducing *additional* primitives to ensure local spinning on DSM machines, as in [61], is not necessary.

We present our generic algorithm by first giving a variant that is designed for CC machines, and by then constructing a DSM variant by applying a simple transformation. This transformation is quite general and can also be applied to the algorithms of T. Anderson [23] and Graunke and Thakkar [38].

By applying our generic algorithm within an arbitration tree, one can easily construct a $\Theta(\log_r N)$ algorithm using any primitive of rank r, where $2 \leq r < N$. For the case $r = \Theta(N)$, this algorithm is clearly asymptotically time-optimal. However, we show that there exists a class of primitives with constant rank for which $\Theta(\log_r N)$ is *not* optimal. We show this by presenting a $\Theta(\log N/ \log\log N)$ algorithm that can be implemented using any primitive that meets an additional condition, which is described next.

In designing a generic algorithm, the key issue to be faced is that of *resetting* a variable that is repeatedly updated by *fetch-and-ϕ* primitive invocations. In our generic algorithm, variables are reset using simple writes. In our $\Theta(\log N/ \log\log N)$ algorithm, such a reset is performed using the *fetch-and-ϕ* primitive itself. That is, this algorithm requires that a *self-resettable* primitive (of rank at least three) be used. Using a self-resettable primitive, the primitive itself can be used to reset a variable that has been updated using that primitive, and hence the resettting operation may return the variable's old value. In our $\Theta(\log N/ \log\log N)$ algorithm, this fact is exploited, with a resulting asymptotic improvement in time complexity for primitives of rank $o(\log N)$. It follows from the $\Omega(\log N/\log\log N)$ lower bound presented in Chapter 5 that this algorithm is time-optimal for certain self-resettable primitives of constant rank.

The rest of this chapter is organized as follows. In Section 8.1, we present needed definitions. Then, in Section 8.2, we present our generic algorithm. The $\Theta(\log N/\log\log N)$ algorithm mentioned above is then presented in Section 8.3. We conclude in Section 8.4. A formal correctness proof for these two algorithms are given in Appendix E and Appendix F, respectively.

8.1 Definitions

We assume the existence of a *generic fetch-and-ϕ* primitive, as defined earlier. We will use "*Vartype*" to denote the type of the accessed variable *var*. (This type is part of the definition of such a primitive.) For example, for a *fetch-and-increment* primitive, *Vartype* would be **integer**, and for a *test-and-set* primitive, it would be **boolean**. In our algorithms, we use $\perp$ to denote the initial value of a variable accessed by a *fetch-and-ϕ* primitive. The choice of $\perp$ is a part of the definition of the given *fetch-and-ϕ* primitive. (For example, if *Vartype* is **boolean**, then $\perp$ would denote either *true* or *false*.) We now define the notion of a "rank," mentioned earlier.

Definition: The *rank* of a *fetch-and-ϕ* primitive is the largest integer r satisfying the following.

For each process p, there exists a constant array $\alpha_p[0..\infty]$ of input values, such that if p performs the following sequence of *fetch-and-ϕ* invocations

$$\textbf{for } i := a_p \textbf{ to } \infty \textbf{ do } \textit{fetch-and-}\phi(v, \alpha_p[i]) \textbf{ od}$$

on a variable v (of type *Vartype*) that is initially $\perp$ (for some choice of $\perp$), where a_p is some integer value, then in any interleaving of these invocations by the N different processes, **(i)** any two invocations among the first $r-1$ by different processes write different values to v, **(ii)** any two *successive* invocations among the first $r-1$ by the same process write different values to v, and **(iii)** of the first r invocations, only the first invocation returns $\perp$.

A *fetch-and-ϕ* primitive has *infinite rank* if the condition above is satisfied for arbitrarily large values of r. $\square$

As our generic algorithm shows, a *fetch-and-ϕ* primitive with rank r has enough power to linearly order r invocations by possibly different processes unambiguously. Note that it is not necessary for the primitive to fully order invocations by the same process, since each process can keep its own execution history.

Examples. An r-bounded *fetch-and-increment* primitive on a variable v with range $0, \ldots, r-1$ is defined by $\phi(\mathit{old}, \mathit{input}) = \min(r-1, \mathit{old}+1)$. (In this primitive, the input parameter is not used, and hence we may simply assume $\alpha_p[j] = \perp$ for all p and j.) If v is initially 0, then any r consecutive invocations on v return distinct values, $0, 1, \ldots, r-1$. Moreover, any further invocation (after the r^{th}) returns $r-1$, which is the same as the return value of the r^{th} invocation. Therefore, an r-bounded *fetch-and-increment* primitive has rank r, and an unbounded *fetch-and-increment* primitive has infinite rank.

For *fetch-and-increment* primitives, the input parameter α is extraneous. However, this is not the case for other primitives. Consider a *fetch-and-store* primitive on a variable with $2N+1$ distinct values ($2N$ pairs $(p, 0)$ and $(p, 1)$, where p is a process, and an additional initial value $\perp$). By defining $\alpha_p[j] = (p, j \textbf{ mod } 2)$, it is easily shown that *fetch-and-store* has infinite rank. (Informally, each process may write the information "this is an (even/odd)-indexed invocation by process p" each time.) It also follows that an unbounded *fetch-and-store* primitive has infinite rank.

Finally, *test-and-set* has rank two: only the first *test-and-set* invocation on a variable initially *false* returns its initial value. *compare-and-swap* also has rank two.

8.2 Constant-time Generic Algorithm

In this section, we present an $O(1)$ mutual exclusion algorithm that uses a generic *fetch-and-ϕ* primitive, which is assumed to have rank at least $2N$. In Section 8.2.1, we present a CC version of this generic algorithm, denoted Algorithm G-CC, which is illustrated in Figure 8.1. Then, in Section 8.2.2, we present a DSM variant, denoted Algorithm G-DSM, which is illustrated in Figure 8.3.

```
shared variables                                                    private variables
    CurrentQueue: 0..1;                                                 idx: 0..1;
    Tail: array[0..1] of Vartype initially ⊥;                           counter: integer;
    Position: array[0..1] of 0..2N − 1 initially 0;                     prev, self, tail: Vartype;
    Signal: array[0..1][Vartype] of boolean initially false;            pos: 0..2N − 1
    Active: array[0..N − 1] of boolean initially false;
    QueueIdx: array[0..N − 1] of (⊥, 0..1)

process p ::    /* 0 ≤ p < N */

while true do
0:  Noncritical Section;
1:  QueueIdx[p] := ⊥;
2:  Active[p] := true;
3:  idx := CurrentQueue;
4:  QueueIdx[p] := idx;
5:  prev := fetch-and-$\phi$(Tail[idx], $\alpha_p$[counter]);
    self := $\phi$(prev, $\alpha_p$[counter]);
    counter := counter + 1;
    if prev ≠ ⊥ then
6:      await Signal[idx][prev];
7:      Signal[idx][prev] := false
    fi;
8:  Entry$_2$(idx);
9:  Critical Section;
10: pos := Position[idx];
11: Position[idx] := pos + 1;
12: Exit$_2$(idx);
13: if (pos < N) ∧ (pos ≠ p) then
14:     await ¬Active[pos] ∨
15:             (QueueIdx[pos] = idx)
    elseif pos = N then
16:     tail := Tail[1 − idx];
17:     Signal[1 − idx][tail] := false;
18:     Tail[1 − idx] := ⊥;
19:     Position[1 − idx] := 0;
20:     CurrentQueue := 1 − idx
    fi;
21: Signal[idx][self] := true;
22: Active[p] := false
od
```

Figure 8.1: ALGORITHM G-CC: Generic *fetch-and-ϕ*-based mutual exclusion algorithm for CC machines.

8.2.1 ALGORITHM G-CC: A Generic Algorithm for CC Machines

When trying to implement a mutual exclusion algorithm using a generic *fetch-and-ϕ* primitive with rank r, the primary problem that arises is that if the primitive is invoked more than r times to access a variable, then it may not be able to provide enough information for processes to order themselves. Therefore, *the algorithm must provide a means of resetting such a variable before it is accessed r times.*

Because we are using a primitive of rank $2N$ in ALGORITHM G-CC, we need to reset a variable accessed by the primitive before it is accessed $2N$ times. We do this by using two "waiting queues," indexed 0 and 1. Associated with each queue j is a "tail pointer," $Tail[j]$. In its entry section, a process enqueues itself onto one of these two queues by using the *fetch-and-ϕ* primitive to update its tail pointer, and waits on its predecessor, if necessary. At any time, one of the queues is designated as the "current"

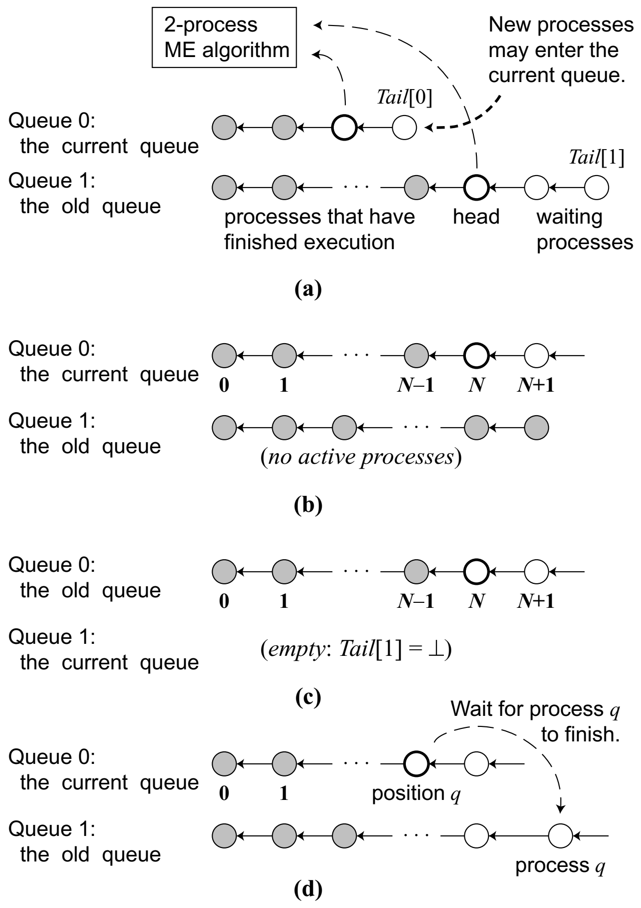


Figure 8.2: The structure of Algorithm G-CC. **(a)** The overall structure. This figure shows a possible state of execution when the current queue is queue 0. (The "finished" processes may be duplicated, because a process may execute its critical section multiple times.) **(b)** A state just before *CurrentQueue* is updated. **(c)** A state just after *CurrentQueue* is updated. **(d)** A process (in its exit section) in the current queue waiting for another in the old queue.

queue, which is indicated by the shared variable *CurrentQueue*. The other queue is called the "old" queue. The algorithm switches between the two queues over time in a way that ensures that each tail pointer is reset before being accessed $2N$ times. We now describe the reset mechanism in detail.

When a process invokes the *Acquire* routine, it determines which queue is the current queue by reading the variable *CurrentQueue* (statement 3 of Figure 8.1), and then enqueues itself onto that queue using the *fetch-and-ϕ* primitive (statement 5). If p is not at the head of its queue ($p.prev \neq \perp$), then it waits until its predecessor in the queue updates the spin variable $Signal[p.idx][p.prev]$ (statement 6), which p then resets (statement 7).

Note that it is possible for a process q to read *CurrentQueue* before another process updates *CurrentQueue* to switch to the other queue. Such a process q will then enqueue itself onto the old queue. Thus, *both* queues may possibly hold waiting processes. To arbitrate between processes in the two queues, an extra two-process mutual exclusion algorithm is used. A process competes in this two-process algorithm after reaching the head of its waiting queue using the routines $\texttt{Entry}_2$ and $\texttt{Exit}_2$, with the index of its queue as a "process identifier" (statements 8 and 12), as illustrated in Figure 8.2(a). Note that this extra two-process algorithm can be implemented from reads and writes in $O(1)$ time using ALGORITHM YA-2 [84], which is described in Section 2.2.2.

As explained above, some process must reset the current queue before it is accessed $2N$ times. To facilitate this, each queue j has an associated shared variable *Position*$[j]$. This variable indicates the relative position of the current head of the queue, starting from 0. For example, in Figure 8.2(a), the head of queue 0 is at position 2, and hence *Position*$[0]$ equals 2. A process in queue j updates *Position*$[j]$ while still effectively in its critical section (statements 10 and 11). Thus, *Position*$[j]$ cannot be concurrently updated by different processes.

A process exchanges the role of the two queues in its exit section if it is at position N in the current queue (statements 16–20). (These statements will be explained in detail shortly.) Figures 8.2(b) and 8.2(c) show the state of the two queues before and after such an exchange. In order to exchange the queues, we must ensure the following property.

Property 1 If a process executes its critical section after having acquired position N of the current queue, then no process is in the old queue.

(A process is considered to be "in" the old queue if it read the index of that queue from *CurrentQueue*. In particular, that process may be yet to update the queue's tail pointer.) Given Property 1, a process at position N may safely reset the old queue and exchange the queues. Property 1 is a direct consequence of the following property.

Property 2 If a process executes its critical section after having acquired position *pos* of the current queue, and if $pos > q$, then process q is not in the old queue.

To maintain Property 2, each process p has two associated variables, *Active*$[p]$ and *QueueIdx*$[p]$, which indicate (respectively) whether process p is active, and if so, which queue it is executing in (statements 1, 2, 4, and 22). If a process p executes at position

$q < N$ in the current queue, then in its exit section, p waits until either q finishes its exit-section execution (*i.e.*, $\mathit{Active}[p] = \mathit{false}$; statement 14) or enters the current queue (statement 15). p thus ensures that process q does not execute in the old queue, and then signals a possible successor (*i.e.*, a process at position $q + 1$ in the current queue) that it is now at the head of the current queue (statement 21). This situation is depicted in Figure 8.2(d).

Although p waits for q, starvation-freedom is guaranteed, because q is in the old queue, and hence makes progress independently of the current queue. Only the current queue is stalled until q finishes execution. (The fact that p may have to wait for a significant duration in its exit section may be a cause for concern. However, such waiting can be eliminated, if process p instructs process q to signal p's successor *after* q finishes its critical section. Thus, p may finish execution without waiting for q. For simplicity, this handshake has not been added to ALGORITHM G-CC.)

We now explain statements 16–20, which are executed in order to exchange the role of the two queues. Without loss of generality, suppose that a process p executes these statements with $p.\mathit{idx} = 0$. (See Figure 8.2(b) and Figure 8.2(c).) Variables $\mathit{Tail}[1]$ and $\mathit{Position}[1]$ are initialized by statements 18 and 19, respectively. In addition, we must ensure that each entry of the $\mathit{Signal}[1][\ldots]$ array is reset to *false*. Note that, if a process q in queue 1 establishes $\mathit{Signal}[1][x] = \mathit{true}$ (where $x = q.\mathit{self}$) by executing statement 21, then its successor r (which spins on $\mathit{Signal}[1][x]$ at statement 6) resets $\mathit{Signal}[1][x]$ by executing statement 7. Thus, by the time p executes statements 16–20, Property 1 ensures that every entry of $\mathit{Signal}[1][\ldots]$ is reset to *false*, *except for the one set by the last process in queue 1.* (Clearly, the last process does not have a successor, so this entry is not reset.)

Property 1 again ensures that this last entry of queue 1 is indicated by $\mathit{Tail}[1]$. Therefore, statements 16 and 17 properly reset this entry and thereby complete the reinitialization of $\mathit{Signal}[1][\ldots]$. Finally, statement 20 exchanges the two queues.

We still must show that using a *fetch-and-*ϕ primitive of rank $2N$ is sufficient. Suppose that process p acquires position N of queue 0 when it is the current queue. We claim that at most $N-1$ processes may be enqueued onto queue 0 after p and before the queues are exchanged again. For a process q to enqueue itself onto queue 0 after p, it must have read the value of $\mathit{CurrentQueue}$ before it was updated by p. For q to enqueue itself a *second* time onto queue 0, it must read $\mathit{CurrentQueue} = 0$ again, *after* $\mathit{CurrentQueue} = 1$ was established by p. This implies that the two queues have been exchanged again. (We remind the reader that, by the explanation above, the queues will not be exchanged again until there are no processes in queue 0.) Thus, after p establishes that queue 1 is current, and while queue 0 continues to be the old queue, at most $N-1$ processes may be enqueued (after p) onto queue 0. Thus, a rank of $2N$ is sufficient.

The busy-waiting loops at statements 6, 14, and 15 in Figure 8.1 are read-only loops in which variables are read that may be updated by a unique process. On a CC machine, each such loop incurs $O(1)$ RMR time complexity. It follows that ALGORITHM G-CC has $O(1)$ RMR time complexity on CC machines.

A detailed correctness proof of ALGORITHM G-CC is given in Appendix E. From the discussion so far, we have the following lemma.

Lemma 8.1 If the underlying *fetch-and-ϕ* primitive has rank at least $2N$, then ALGORITHM G-CC is a correct, starvation-free mutual exclusion algorithm with $O(1)$ RMR time complexity in CC machines. □

8.2.2 ALGORITHM G-DSM: A Generic Algorithm for DSM Machines

We now explain how to convert ALGORITHM G-CC into ALGORITHM G-DSM, which is illustrated in Figure 8.3. The key idea of this conversion is a simple transformation of each busy-waiting loop, which we examine here in isolation. In ALGORITHM G-CC, all busy waiting is by means of statements of the form "**await** B," where B is some boolean condition. Moreover, if a process p is waiting for condition B to hold, then there is a unique process that can establish B, and once B is established, it remains true, until p's "**await** B" statement terminates.

In ALGORITHM G-DSM, each statement of the form "**await** B" has been replaced by the code fragment on the left below (see statements 10–17 and 25–33 in Figure 8.3), and each statement of the form "$B := true$" by the code fragment on the right (see statements 4–8, 39–43, and 44–48).

```
a: Entry$_2(\mathcal{J}, 0)$;
b:     $flag := B$;
c:     $Waiter[\mathcal{J}] :=$ if $flag$ then $\perp$ else $p$;
d:     $Spin[p] := false$;
e: Exit$_2(\mathcal{J}, 0)$;
f: if $\neg flag$ then
g:     await $Spin[p]$;
h:     $Waiter[\mathcal{J}] := \perp$
   fi
```

```
i: Entry$_2(\mathcal{J}, 1)$;
j:     $B := true$;
k:     $next := Waiter[\mathcal{J}]$;
l: Exit$_2(\mathcal{J}, 1)$;
m: if $next \neq \perp$ then $Spin[next] := true$ fi
```

```
/* all variable declarations are as defined in Figure 8.1 except as noted here */
shared variables
    Waiter1: array[0..N − 1] of (⊥, 0..N − 1);
    Waiter2: array[0..1][Vartype] of (⊥, 0..N − 1);
    Spin: array[0..N − 1] of boolean initially false

private variables
    next: (⊥, 0..N − 1);
    flag: boolean;
    q: 0..N − 1

process p ::    /* 0 ≤ p < N */
while true do
0:      Noncritical Section;
1:      QueueIdx[p] := ⊥;
2:      Active[p] := true;
3:      idx := CurrentQueue;
**4:**  Entry_2(p, 1);
**5:**      QueueIdx[p] := idx;
**6:**      q := Waiter1[p];
**7:**  Exit_2(p, 1);
**8:**  if q ≠ ⊥ then Spin[q] := true fi;
9:      prev := fetch-and-φ(Tail[idx], α_p[counter]);
        self := φ(prev, α_p[counter]);
        counter := counter + 1;
        if prev ≠ ⊥ then
**10:**     Entry_2((idx, prev), 0);
**11:**         flag := Signal[idx][prev];
**12:**         Waiter2[idx][prev] :=
                                    if flag then ⊥ else p;
**13:**         Spin[p] := false;
**14:**     Exit_2((idx, prev), 0);
**15:**     if ¬flag then
**16:**         await Spin[p];
**17:**         Waiter2[idx][prev] := ⊥
            fi;
18:         Signal[idx][prev] := false
        fi;
19:     Entry_2(idx)
20:     Critical Section;
21:     pos := Position[idx];
22:     Position[idx] := pos + 1;
23:     Exit_2(idx);
24:     if (pos < N) ∧ (pos ≠ p) then
            q := pos;
**25:**     Entry_2(q, 0);
**26:**         flag :=¬Active[q] ∨
**27:**                   (QueueIdx[q] = idx);
**28:**         Waiter1[q] :=
                                if flag then ⊥ else p;
**29:**         Spin[p] := false;
**30:**     Exit_2(q, 0);
**31:**     if ¬flag then
**32:**         await Spin[p];
**33:**         Waiter1[q] := ⊥
            fi
        elseif pos = N then
34:         temp := Tail[1 − idx];
35:         Signal[1 − idx][temp] := false;
36:         Tail[1 − idx] := ⊥;
37:         Position[1 − idx] := 0;
38:         CurrentQueue := 1 − idx
        fi;
**39:**  Entry_2((idx, self), 1);
**40:**      Signal[idx][self] := true;
**41:**      next := Waiter2[idx][self];
**42:**  Exit_2((idx, self), 1);
**43:**  if next ≠ ⊥ then Spin[next] := true fi;
**44:**  Entry_2(p, 1);
**45:**      Active[p] := false;
**46:**      next := Waiter1[p];
**47:**  Exit_2(p, 1);
**48:**  if next ≠ ⊥ then Spin[next] := true fi
od
```

Figure 8.3: ALGORITHM G-DSM: Generic *fetch-and-ϕ*-based mutual exclusion algorithm for DSM machines. Statements different from Figure 8.1 are shown with **boldface** line numbers.

The variable $Waiter[\mathcal{J}]$ is assumed to be initially $\perp$, and $Spin[p]$ is a spin variable used exclusively by process p (and, hence, it can be stored in memory local to p). $\texttt{Entry}_2$ and $\texttt{Exit}_2$ represent an instance of a two-process mutual exclusion algorithm, indexed by $\mathcal{J}$. To see that this transformation is correct, assume that a process p executes lines a–h while another process q executes lines i–m. Since lines b–d and j–k execute within a critical section, lines b–d precede lines j–k, or *vice versa*. If b–d precede j–k, and if $B = \mathit{false}$ holds before the execution of b–d, then p assigns $Waiter[\mathcal{J}] := p$ at line c, and initializes its spin variable at line d. Process q subsequently reads $Waiter[\mathcal{J}] = p$ at line k, and establishes $Spin[p] = \mathit{true}$ at line m, which ensures that p is not blocked. On the other hand, if lines j–k precede lines b–d, then process q reads $Waiter[\mathcal{J}] = \perp$ (the initial value) at line k, and does not update any spin variable at line m. Since process p executes line b after q executes line j, p preserves $Waiter[\mathcal{J}] = \perp$, and does not execute lines g and h. Given the correctness of this transformation, we have the following.

Lemma 8.2 If the underlying *fetch-and-ϕ* primitive has rank at least $2N$, then Algorithm G-DSM is a correct, starvation-free mutual exclusion algorithm with $O(1)$ RMR time complexity in DSM machines. □

The transformation above also can be applied to the algorithms of T. Anderson [23] and Graunke and Thakkar [38], which are described in Section 2.2.1. In each case, the two-process mutual algorithm actually can be avoided by utilizing the specific *fetch-and-ϕ* primitive used (*fetch-and-increment* and *fetch-and-store*, respectively).

If we have a *fetch-and-ϕ* primitive with rank r $(4 \leq r < 2N)$, then we can arrange instances of Algorithm G-DSM in an arbitration tree, where each process is statically assigned a leaf node and each non-leaf node consists of an $\lfloor r/2 \rfloor$-process mutual exclusion algorithm, implemented using Algorithm G-DSM. Because this arbitration tree is of $\Theta(\log_r N)$ height, we have the following theorem. (Note that for $r = 2$ or 3, a $\Theta(\log_r N)$ algorithm is possible without even using the *fetch-and-ϕ* primitive; see Algorithm YA-N in Section 2.2.2.)

Theorem 8.1 *Using any fetch-and-ϕ primitive of rank $r \geq 2$, starvation-free mutual exclusion can be implemented with $\Theta(\max(1, \log_r N))$ RMR time complexity on either CC or DSM machines.* □

We can combine this arbitration-tree algorithm with our adaptive algorithm (Algorithm A-LS) presented in Chapter 4. In particular, the height of the renaming tree is changed from $\log N$ to $\log_r N$, and $\texttt{Entry}_N/\texttt{Exit}_N$ calls are replaced by the *fetch-and-ϕ*-based arbitration-tree algorithm. Therefore, we have the following theorem.

Theorem 8.2 *Using any fetch-and-ϕ primitive of rank $r \geq 2$, starvation-free adaptive mutual exclusion can be implemented with $O(\min(k, \max(1, \log_r N)))$ RMR time complexity on either CC or DSM machines, where k is point contention.* □

8.3 $\Theta(\log N/\log\log N)$ Algorithms

The time-complexity bound in Theorem 8.1 is clearly tight for $r = \Theta(N)$. In this section, we show that for some primitives of rank $r = o(\log N)$, it is *not* tight, provided that r is at least three. This follows from Algorithm T, shown in Figure 8.9, which has $\Theta(\log N/\log\log N)$ RMR time complexity on both DSM and CC machines. In this algorithm, it is assumed that the *fetch-and-ϕ* primitive used has a rank of at least three, and is "self-resettable," as defined below.

Definition: A *fetch-and-ϕ* primitive with rank r is *self-resettable* if the following hold.

- Let $\alpha_p[0..k_p - 1]$ be defined as in the definition of rank, and let each process execute the **for** loop shown in that definition. Then, in any interleaving of an *arbitrary number* of these *fetch-and-ϕ* primitive invocations by the N different processes, only the first invocation returns $\perp$.

- For each $\alpha_p[i]$, there is an associated value $\beta_p[i]$ such that $\phi(\phi(\perp, \alpha_p[i]), \beta_p[i]) = \perp$. That is, if the invocation of the *fetch-and-ϕ* primitive on v by process p returns $\perp$, and if no other process accesses v, then p may *reset* the variable by invoking the primitive again with a "reset" parameter. □

Recall that in the generic algorithms of the previous section, devising a way of resetting the *Tail* variables was the key problem to be addressed. Because we could assume so little of the semantics of the *fetch-and-ϕ* primitive being used, simple write operations were used to reset these variables. If a self-resettable *fetch-and-ϕ* primitive is available, then that primitive itself can be used to perform such a reset.

```
type $NodeType$ = record $winner$, $waiter$: $(0..N-1, \perp)$ end;
/* if $winner = \perp$, then $waiter = \perp$ also holds. */

process $p$ ::    /* $0 \le p < N$ */

function $AcquireNode(t : NodeType)$: (WINNER, PRIMARY_WAITER, SECONDARY_WAITER)
/* atomically do the following */
    if $t = (\perp, \perp)$ then
        $t := (p, \perp)$;  return WINNER
    elseif $t.waiter = \perp$ then
        $t.waiter := p$;  return PRIMARY_WAITER
    else
        return SECONDARY_WAITER
    fi

function $ReleaseNode(t : NodeType)$: (SUCCESS, FAIL)
/* atomically do the following */
    if $t.winner \neq p$ then
        /* error: should not happen */
    elseif $t = (p, \perp)$ then
        $t := (\perp, \perp)$;  return SUCCESS
    else
        return FAIL
    fi
```

Figure 8.4: Definitions of *NodeType*, *AcquireNode*, and *ReleaseNode*. Note that *AcquireNode* and *ReleaseNode* are assumed to execute atomically.

8.3.1 Algorithm T0: A Simple Tree Algorithm

As a stepping stone toward Algorithm T, we present a simpler algorithm, Algorithm T0, with a similar structure. Algorithm T0, which is shown in Figure 8.5, uses an arbitration tree, each node n of which is represented by a "local variable" $Lock[n]$ of type *NodeType*. Such a variable can hold up to two process identifiers and is accessible ty two atomic operations, *AcquireNode* and *ReleaseNode*, as shown in Figure 8.4, in addition to ordinary read and write operations. (Later, in Algorithm T, these operations are replaced by invocations of an arbitrary self-resettable *fetch-and-ϕ* primitive of rank at least three.)

Informally, a value of $(\perp, \perp)$ represents an available node; $(p, \perp)$, where $p \neq \perp$, represents a situation in which process p has acquired the node and no other process has since accessed that node; (p, q), where $p \neq \perp$ and $q \neq \perp$, represents a situation in which p has acquired the node and another process q is waiting at that node (perhaps along with some other processes).

```
shared variables
    Lock: array[1..MAX_NODE] of NodeType initially $(\perp, \perp)$;
    Spin: array[$0..N-1$] of boolean;
    WaitingQueue: serial waiting queue initially empty;
    Promoted: $(\perp, 0..N-1)$ initially $\perp$

private variables
    lev, break_level: 0..MAX_LEVEL;
    n, child: 1..MAX_NODE;
    proc: $\perp, 0..N-1$;
    side: 0..1

process p ::    /* $0 \leq p < N$ */

while true do
0:  Noncritical Section;

1:  Spin[p] := false;
2:  AcquireNode(Lock[Node(p, MAX_LEVEL)]);
    /* automatically acquire its leaf node */

    lev := MAX_LEVEL $- 1$;
    break_level := 0;
    repeat
        n := Node(p, lev);
3:      if AcquireNode(Lock[n]) =
              WINNER then
            lev := lev $- 1$
        else
            break_level := lev
        fi
    until (lev = 0) $\vee$ (break_level > 0);
    side := if break_level = 0 then 0
                                  /* normal entry */
              else 1;
                                  /* promoted entry */

4:  if side = 1 then
        await Spin[p]
                        /* wait until promoted */
    fi;
5:  Entry$_2$(side);

6:  Critical Section;

7:  Wait();           /* wait at the barrier */
8:  Exit$_2$(side);

    for lev := break_level + 1 to
          MAX_LEVEL $- 1$ do
        /* reopen each node p has acquired */
        n := Node(p, lev);
9:      if Lock[n].winner = p then
10:         if ReleaseNode(Lock[n]) = FAIL then
11:             Enqueue(WaitingQueue,
                        Lock[n].waiter);
12:             Lock[n] := $(\perp, \perp)$
        fi fi
    od;

13: n := if side = 1 then Node(p, break_level)
                              /* promoted at node n */
          else 1;             /* the root node */
14: if Lock[n].waiter = p then
15:     proc := Lock[n].winner;
        if n = 1 then
16:         Lock[n] := (proc, $\perp$)
        else
17:         Lock[n] := $(\perp, \perp)$;
18:         Enqueue(WaitingQueue, proc)
    fi fi;
    for each child := (a child of n) do
19:     proc := Lock[child].winner;
20:     if (proc $\neq \perp$) then
            Enqueue(WaitingQueue, proc) fi
    od;

21: ReleaseNode(Lock[Node(p, MAX_LEVEL)]);
                              /* reopen its leaf node */
22: Remove(WaitingQueue, p);
23: proc := Promoted;
    if (proc = p) $\vee$ (proc = $\perp$) then
24:     proc := Dequeue(WaitingQueue);
25:     Promoted := proc;
26:     if proc $\neq \perp$ then Spin[proc] := true fi
    fi;

27: Signal()                  /* open the barrier */
od
```

Figure 8.5: ALGORITHM T0: A tree-structured algorithm using a *NodeType* object.

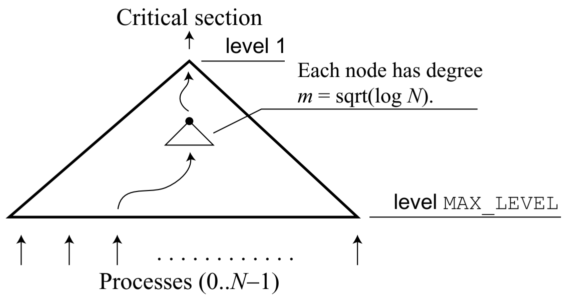


Figure 8.6: Arbitration tree of Algorithms T0 and T.

Arbitration tree and waiting queue. The structure of the arbitration tree is illustrated in Figure 8.6. The tree is of degree $m = \sqrt{\log N}$. Each process is statically assigned to a leaf node, which is at level MAX_LEVEL. (The root is at level 1.) Since the tree has N leaf nodes, MAX_LEVEL $= \Theta(\log_m N) = \Theta(\log N/\log\log N)$.

To enter its critical section, a process p traverses the path from its leaf up to the root and attempts to acquire each node on this path. If p acquires the root node, then it may enter its critical section. As explained shortly, p may also be "promoted" to its critical section while still executing within the tree. (In that case, p may have acquired only some of the nodes on its path.) In either case, upon exiting its critical section, p traverses its path in reverse, releasing each node it has acquired.

In addition to the arbitration tree, a serial waiting queue, *WaitingQueue*, is used. This queue is accessed by a process only within its exit section. A "barrier" mechanism is used that ensures that multiple processes do not execute their exit sections concurrently (statements 7 and 27 of Figure 8.5). As a result, the waiting queue can be implemented as a sequential data structure. It is accessible by the usual *Enqueue* and *Dequeue* operations, and also an operation *Remove*(*WaitingQueue*, p), which removes process p from inside the queue, if present; it is straightforward to implement each of these operations in $O(1)$ time. When a process p, inside its exit section, discovers another waiting process q, p adds q to the waiting queue. In addition, p dequeues a process r from the queue (if the queue is nonempty), and "promotes" r to its critical section. (Similar mechanism is also used in Section 5.4, and in [24, 30, 41, 48]; see the remark on page 139.)

Arbitration at a node. As mentioned above, associated with each (non-leaf) node n is a "lock variable" *Lock*[n], which represents the state of that node. The structure of such a node is illustrated in Figure 8.7. In its entry section, a process p may try to

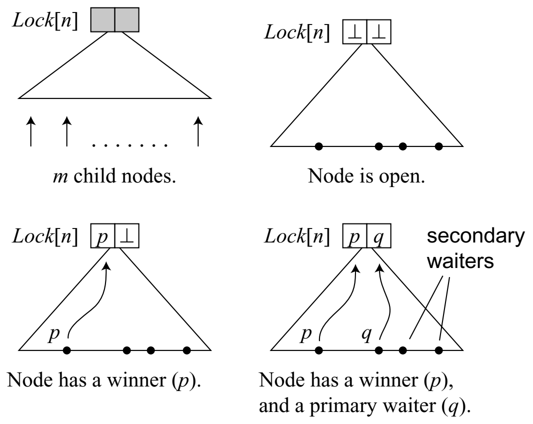


Figure 8.7: Structure of a node used in Algorithm T0.

acquire node n only if it has already acquired some child of n. In order to acquire node n, p executes $\mathit{AcquireNode}(\mathit{Lock}[n])$. Assume that the old value of $\mathit{Lock}[n]$ is (q, r). There are three possibilities to consider.

- If $q = \perp$ holds, then p has established $\mathit{Lock}[n] = (p, \perp)$ and has acquired node n. In this case, p becomes the *winner* of node n, and proceeds to the next level of the tree. (Note that $r = \perp$ holds by definition in this case.)

- If $q \neq \perp$ and $r = \perp$ hold, then p has established $\mathit{Lock}[n] = (q, p)$, in which case it becomes the *primary waiter* at node n. In this case, p stops at node n and waits until it is "promoted" to its critical section by some other process.

- Otherwise, the value of $\mathit{Lock}[n]$ is not changed, in which case p is a *secondary waiter* at node n. In this case, p also waits at node n until it is promoted.

Next, consider the behavior of a process p in its exit section. There are two possibilities to consider, depending on p's execution history in its entry section.

- If p acquired node n in its entry section, then p has established $\mathit{Lock}[n] = (p, \perp)$. In this case, p tries to release node n by executing $\mathit{ReleaseNode}(\mathit{Lock}[n])$. If no other process has updated $\mathit{Lock}[n]$ between p's executions of $\mathit{AcquireNode}(\mathit{Lock}[n])$ and $\mathit{ReleaseNode}(\mathit{Lock}[n])$, then node n is successfully released (*i.e.*, $\mathit{Lock}[n]$ transits to $(\perp, \perp)$). In this case, p descends the tree and continues to release other nodes it has acquired.

  On the other hand, if some other process has updated $\mathit{Lock}[n]$ between p's two invocations, then let q be the first such process. As explained above, q must have changed $Lock[n]$ from $(p, \perp)$ to (p, q), thus designating itself as the primary waiter at node n. In this case, p adds q to the waiting queue. (Note that p does *not* enqueue any secondary waiters, *i.e.*, processes that accessed $Lock[n]$ after q.) Process p then releases node n by writing (*not* via calling *ReleaseNode*) $Lock[n] := (\perp, \perp)$, and descends the tree.

- If p was promoted at node n, then p has *not* acquired node n, and hence is not responsible for releasing node n. Instead, p examines every child of node n (specifically, $Lock[child]$, where *child* is a child of n) to determine if any "secondary waiters" at node n exist. p adds such processes to the waiting queue, and descends the tree.

The algorithm uses an additional mechanism that ensures the following properties, as explained shortly.

Property 3 If a process p acquires a non-root node $n > 1$, and if another process q later becomes the primary waiter of node n, then q examines every child of node n *after* node n is released by p or by some other process on behalf of p (see below).

Property 4 If a process p acquires the root node (node 1), then p examines every child of node 1 *after* node 1 is released by p.

(The reason that we need a separate property for the root node is that the winner of the root node may immediately enter its critical section. Therefore, in order to maintain the Exclusion property, the root node can be released only by its winner.[1] On the other hand, non-root nodes can be released by some other process, as explained later.)

Assuming these properties, we can easily show that each process eventually either acquires the root, or is added to the waiting queue by some other process. In particular, at node n, the winner always proceeds to the next level, and the primary waiter q is eventually enqueued by the winner or by some other process. (The latter could happen if waiting processes on q's path lower in the tree are promoted.) Thus, we only have to show that a secondary waiter is eventually enqueued. In order for a process r to become a secondary waiter at node n, it must first acquire a child node n' of n, and then execute $AcquireNode(Lock[n])$ while $Lock = (p, q)$ holds, for some winner p and

[1]In the preliminary version of the work presented in this chapter [19], the root node is not considered as a special case. We later found that this can lead to a violation of the Exclusion property.

primary waiter q. If $n > 1$, then Property 3 guarantees that q has yet to examine the child nodes of n; if $n = 1$ (*i.e.*, n is the root node), then Property 4 guarantees that p has yet to examine the child nodes of n. Therefore, in either case, p or q eventually examines node n', and adds r to the waiting queue (if it has not already been added by some other process).

Finally, since the waiting queue is checked every time a process executes its exit section, it follows that the algorithm is starvation-free.

As explained above, processes exiting the arbitration tree form two groups: the promoted processes and the non-promoted processes (*i.e.*, those that successfully acquire the root). To arbitrate between these two groups, an additional two-process mutual exclusion algorithm is used (statements 5 and 8).

Further details. Having explained the basic structure of the algorithm, we now present a more detailed overview. We begin by considering the shared variables used in the algorithm, which are listed in Figure 8.5. *Lock* and *WaitingQueue* have already been explained. *Spin*[p] is a dedicated spin variable for process p. *Promoted* is used to hold the identity of any promoted process. This variable is used to ensure that multiple processes are not promoted concurrently, which is required in order to ensure that the additional two-process mutual exclusion algorithm is accessed by only one promoted process at a time.

We now consider ALGORITHM T0 in some detail. A process p in its entry section first initializes its spin variable (statement 1), and automatically acquires its leaf node (statement 2). It then ascends the arbitration tree (the **repeat** loop at statement 3). Function Node(p, *lev*) is used to return the index of the node at level *lev* in p's path. Process p tries to acquire each node it visits by invoking *AcquireNode* (statement 3). If it succeeds, then it ascends to the next level; otherwise, it finishes accessing the arbitration tree. The private variable *break_level* stores the level at which p exited the **repeat** loop.

After exiting the loop, the private variable *side* is assigned the value of 0 if p successfully acquired the root node, and 1 otherwise. If *side* $= 0$, then p executes the two-process entry section using "0" as a process identifier (statement 5). Otherwise, p spins until it is promoted by some other process (statement 4), and executes the two-process entry section using "1" as a process identifier (statement 5).

In its exit section, p waits until the barrier is opened (statement 7) and then executes the two-process exit section (statement 8). The barrier is specified by two procedures

`Wait` and `Signal`, which ensure that p waits at statement 7 if another process is executing within statements 8–26. Because `Wait` is invoked within a critical section, it is straightforward to implement these procedures in $O(1)$ time. In CC machines, `Wait` can be defined as "**await** *Flag*; *Flag* := *false*" and `Signal` as "*Flag* := *true*," where *Flag* is a shared boolean variable. In DSM machines, a slightly more complicated implementation is required, as explained shortly.

Process p then tries to reopen each non-leaf node that it acquired in its entry section (statements 9–12). For each such node n, p checks if it is still the winner (statement 9); this may not be the case, if the primary waiter at node n executed statements 15–18 before p entered its critical section, as explained shortly. If p is indeed the winner at node n, then it tries to reopen node n (statement 10). p may fail to reopen node n only if node n has a primary waiter, in which case p enqueues the waiter and reopens the node using an ordinary write (statements 11 and 12).

Statements 13–18 are executed in order to maintain Properties 3 and 4. After that, statements 19 and 20 are executed to promote the secondary waiters. We now examine these statements in detail by considering three cases.

- First, assume that p was promoted at node $k > 1$ (*i.e.*, $break\_level > 1 \wedge side = 1$). In this case, statement 13 assigns $n := k$. p then examines $Lock[k]$ (statement 14). If p finds $Lock[k].waiter = p$ at statement 14, then p is the primary waiter at k, and was promoted *before* the winner of node k (say, q, given by $Lock[k].winner$) entered its critical section. This can happen because p may actually have been promoted by a primary waiter at a lower level. In this case, p resets node k in place of process q, and adds q to the waiting queue (statements 15, 17, and 18).

  Note that node k may be thus reset even before q finishes its entry section. If yet another process r subsequently acquires node k, then we may simultaneously have two processes q and r, each of which "thinks" that it is the winner of node k. However, since k is not a root node, this does not violate the Exclusion property. Moreover, starvation-freedom is ensured as follows. Process q was already added to the waiting queue by p. Since $Lock[k].winner = r$ holds after r becomes the new winner of node k, r either acquires the parent node k' of k and continue progress, or is eventually discovered by the winner of node k' (or by some other process). Therefore, both q and r eventually enter their critical sections.

After adding q to the waiting queue, p adds any process that has acquired a child node of k to the waiting queue (statements 19 and 20). Note that statements 14–18 ensure that $Lock[k]$ is released at least once before statements 19 and 20 are executed, thus ensuring that Property 3 holds.

- Second, assume that p successfully acquired the root node (*i.e.*, $break\_level = 0 \land side = 0$). In this case, statement 13 assigns $n := 1$. Since p is the winner of the root node, clearly it is not the primary waiter of the root node. Therefore, p does not execute statements 15–18. Note that p has already reopened the root node (together with any other node it has acquired) by executing the **for** loop at statements 9–12.

  After that, p adds any process that has acquired a child node of the root node to the waiting queue (statements 19 and 20). Note that statements 9–12 ensure that $Lock[1]$ is released once before statements 19 and 20 are executed, thus ensuring that Property 4 holds.

- Finally, assume that p was promoted at the root node (*i.e.*, $break\_level = 1 \land side = 1$). In this case, statement 13 assigns $n := 1$. p then examines $Lock[1]$ (statement 14). If p finds $Lock[1].waiter = p$ at statement 14, then p is the primary waiter at the root node, and was promoted *before* the winner (say, q) of the root node entered its critical section. In this case, p resets only $Lock[1].waiter$ (statements 15 and 16). Hence, it is maintained that the root node is reset only by its winner.

  After that, p executes statements 19 and 20. (This is in fact unnecessary, since q will later execute these statements.)

Finally, p resets its leaf node (statement 21), makes sure that it is not contained in the waiting queue (statement 22), and checks if there is any unfinished promoted process (statement 23). If not, then p dequeues and promotes a process from the waiting queue (if one exists) (statements 24–26). As a last step, p opens the barrier (statement 27).

The barrier mechanism. We now explain how to implement the barrier mechanism for DSM systems, provided that we have a *fetch-and-ϕ* primitive of rank at least two. Procedures `Wait` and `Signal` can be implemented by the code fragments on the left and right below, respectively. (b, *counter*, and *next* are private variables.)

```
a:  $Waiter := p$;
b:  $Spin'[p] := false$;
c:  $b := fetch\text{-}and\text{-}\phi(B, \alpha_p[counter])$;
    $counter := counter + 1$;
    if $b = \bot$ then
d:      await $Spin'[p]$
    fi;
e:  $B := \bot$;
f:  $Waiter := \bot$
```

```
g:  $b := fetch\text{-}and\text{-}\phi(B, \alpha_p[counter])$;
    $counter := counter + 1$;
    if $b \neq \bot$ then
h:      $next := Waiter$;
i:      $Spin'[next] := true$
    fi
```

If a process p executes `Wait`, then it establishes $B = \bot$ at line e. Thus, if $B \neq \bot$ holds, then either the barrier is open (*i.e.*, p has executed line g), or there is a process that has executed line c and is waiting at line d. In the latter case, *Waiter* indicates the current waiting process. $Spin'[p]$ is a spin variable used exclusively by process p (and, hence, it can be stored in memory local to p). Recall that there exist at most one process that may execute within a–e (respectively, g–i) at any time. In particular, lines a–e (statement 7 of Figure 8.5) is protected by $\texttt{Entry}_2$ and $\texttt{Exit}_2$ calls, and lines g–i (statement 27 of Figure 8.5) is protected by the barrier itself. Therefore, it is straightforward to establish the correctness of this implementation.

Time complexity. In order to compute the time complexity of Algorithm T0, note that `MAX_LEVEL` $= \Theta(\log N/\log\log N)$ holds. Therefore, the loops at statement 3 and statements 9–12 iterate $O(\log N/\log\log N)$ times each. Since the arbitration tree has degree $\Theta(\sqrt{\log N})$, the **for** loop in statements 19–20 iterates $\Theta(\sqrt{\log N})$ times, which is asymptotically dominated by $\Theta(\log N/\log\log N)$. Finally, the loop in statement 7 spins on a local spin variable, and hence incurs $O(1)$ RMR time complexity. It follows that Algorithm T0 has $\Theta(\log N/\log\log N)$ RMR time complexity on both DSM and CC machines.

8.3.2 Algorithm T: A Generic Tree Algorithm

We now explain the differences between Algorithm T0 and Algorithm T, which is shown in Figure 8.9.

In order to hide certain low-level details in Algorithm T, we will assume the availability of two operations, *fetch-and-update* and *fetch-and-reset*, illustrated below. A *fetch-and-update* operation on a variable v invokes the *fetch-and-ϕ* primitive being used with the parameter $\alpha_p[counter_v]$ (where $counter_v$ is a private counter variable associated with v), increments $counter_v$, and returns the old value of v (*i.e.*, the return value of the *fetch-and-ϕ* primitive) and the new value of v (which can be determined by $\phi(v, \alpha_p[counter_v])$). A *fetch-and-reset* operation on a variable v invokes the *fetch-and-ϕ* primitive with the parameter $\beta_p[counter_v]$, and also returns the old and new values of v.

```
. process $p$ :

function $fetch\text{-}and\text{-}update(v\colon\ Vartype)\colon\ (Vartype,\ Vartype)$
    $counter_v := counter_v + 1;$
    $temp := fetch\text{-}and\text{-}\phi(v,\ \alpha_p[counter_v]);$
    $retval := (temp,\ \phi(temp,\ \alpha_p[counter_v]));$
    return $retval$

function $fetch\text{-}and\text{-}reset(v\colon\ Vartype)\colon\ (Vartype,\ Vartype)$
    $temp := fetch\text{-}and\text{-}\phi(v,\ \beta_p[counter_v]);$
    $retval := (temp,\ \phi(temp,\ \beta_p[counter_v]));$
    return $retval$
```

In ALGORITHM T, each lock variable is accessed by the *fetch-and-update* and *fetch-and-reset* operations, instead of the *AcquireNode* and *ReleaseNode* operations in Figure 8.4. Each such variable is assumed to have a type (*Vartype*) that is consistent with the given *fetch-and-ϕ* primitive being used, and is initially $\perp$. The main problem associated with the use of a generic *fetch-and-ϕ* primitive is that we cannot use the same variable as both a lock variable and as a variable for storing process identifiers. In particular, even if a process p performs a successful *fetch-and-update*(v) operation, the value written to v may be completely arbitrary; another process q may not be able to discover the winning process (*i.e.*, p) by reading v. Therefore, we need a pair of variables, one for each purpose.

Another problem is that, in its exit section, a winner p (at node n) may fail to discover the primary waiter. To see why this is so, consider the following scenario: $Lock[n]$ is initially $\perp$; p acquires node n (*via* a *fetch-and-update* invocation), thus writing v_1 to $Lock[n]$; another process q accesses $Lock[n]$, writes v_2, and becomes the primary waiter; yet another process r accesses $Lock[n]$, and writes v_1. (This is allowed because the primitive may have rank three.) Thus, process p cannot detect q and r by reading $Lock[n]$.

In order to solve this problem, note that such a situation may arise only if there are multiple waiters (q and r in this case). We exploit this fact by supplying *two* lock variables $Lock[n][0]$ and $Lock[n][1]$ to each node n. A separate variable $WaiterLock[n]$ is used to elect a primary waiter. Thus, we can design the entry section of each node as follows.

```
shared variables
    Lock: array[1..MAX_NODE][0..1] of Vartype initially ⊥;
    WaiterLock: array[1..MAX_NODE] of Vartype initially ⊥;
    Winner: array[1..MAX_NODE][0..1] of (⊥, 0..N − 1) initially ⊥;
    Waiter: array[1..MAX_NODE] of (⊥, 0..N − 1) initially ⊥;
    Spin: array[0..N − 1] of boolean;
    WaitingQueue: serial waiting queue initially empty;
    Promoted: (⊥, 0..N − 1) initially ⊥

private variables
    lev, break_level: 0..MAX_LEVEL;
    n, child: 1..MAX_NODE;
    proc: ⊥, 0..N − 1;
    result: (PRIMARY_WINNER, PRIMARY_WAITER,
             SECONDARY_WINNER, SECONDARY_WAITER);
    prev, new: Vartype;
    lock: array[1..MAX_LEVEL] of Vartype;
    side: 0..2;
    i: 0..1
```

Figure 8.8: Variables used in Algorithm T.

- First, a process executes *fetch-and-update*(*Lock*[n][0]) in order to become the *primary winner* at node n.
- If it fails to become the primary winner, then it executes *fetch-and-update*(*WaiterLock*[n]) in order to become the primary waiter.
- If it still fails to become the primary waiter, then it executes *fetch-and-update*(*Lock*[n][1]) in order to become the *secondary winner* at node n.
- Finally, if it fails to become the secondary winner, then it becomes a *secondary waiter*.

A process ascends the tree if it becomes either the primary winner or the secondary winner. Thus, now two processes can ascend the tree at each node. Note that a process may become the secondary winner only if it fails to become the primary waiter, *i.e.*, only if there already exists a primary waiter. Therefore, if the primary winner (in its exit section) fails to detect the primary waiter, then some process must become a secondary winner that *knows* that there exists a primary waiter. Thus, in this case, the primary winner may safely descend the arbitration tree without promoting the primary waiter; the primary waiter is eventually promoted by the secondary winner.

We now explain the structure of Algorithm T in detail.

```
process $p$ ::    /* $0 \le p < N$ */
while $true$ do
0:   Noncritical Section;
1:   $Spin[p] := false$;
2:   $Winner[\mathsf{Node}(p, \texttt{MAX\_LEVEL})][0] := p$;      /* automatically acquire its leaf node */
     $lev,\ break\_level := \texttt{MAX\_LEVEL} - 1,\ 0$;
     repeat
3:       $result := AcquireNode(lev)$;
         if $(result = \texttt{PRIMARY\_WINNER}) \lor (result = \texttt{SECONDARY\_WINNER})$ then
             $lev := lev - 1$
         else
             $break\_level := lev$
         fi
     until $(lev = 0) \lor (break\_level > 0)$;
     $side :=$ if $result = \texttt{PRIMARY\_WINNER}$ then 0
             elseif $result = \texttt{SECONDARY\_WINNER}$ then 1
             else 2;
4:   if $side = 2$ then await $Spin[p]$ fi;                    /* wait until promoted */
5:   $\texttt{Entry}_3(side)$;
6:   Critical Section;
7:   $\texttt{Wait}()$;                                          /* wait at the barrier */
8:   $\texttt{Exit}_3(side)$;
     for $lev := break\_level + 1$ to $\texttt{MAX\_LEVEL} - 1$ do
         /* reopen each non-leaf node $p$ has acquired */
         $n := \mathsf{Node}(p, lev)$;
9:       if $Winner[n][0] = p$ then                             /* primary winner */
10:          $Winner[n][0] := \perp$;
11:          $(prev, new) := fetch\text{-}and\text{-}reset(Lock[n][0])$;
             if $(n > 1) \land (prev \ne lock[lev])$ then
12:              repeat $proc := Waiter[n]$ until $proc \ne \perp$;
13:              $Enqueue(WaitingQueue, proc)$
             fi;
14:          if $new \ne \perp$ then $Lock[n][0] := \perp$ fi
15:      elseif $Winner[n][1] = p$ then                         /* secondary winner */
16:          $Winner[n][1] := \perp$;
17:          $Lock[n][1] := \perp$;
18:          if $WaiterLock[n] \ne \perp$ then
19:              repeat $proc := Waiter[n]$ until $proc \ne \perp$;
20:              $Enqueue(WaitingQueue, proc)$
         fi fi
     od;
```

Figure 8.9: Algorithm T. (Continued on the next page.)

```
21: $n$ := **if** $side = 2$ **then** Node($p$, $break\_level$)     /* promoted at node $n$ */
            **else** 1;    /* the root node */
22: **if** $Waiter[n] = p$ **then**                                        /* primary waiter */
23:     $Waiter[n] := \perp$;
24:     $WaiterLock[n] := \perp$
    **fi**;
25: **if** $(n > 1) \wedge (Lock[n][0] \neq \perp)$ **then**
26:     **repeat** $proc := Winner[n][0]$ **until** $proc \neq \perp$;
27:     $Winner[n][0] := \perp$;
28:     $Lock[n][0] := \perp$;
29:     $Enqueue(WaitingQueue, proc)$
    **fi**;
    **for each** $child$ := (a child of $n$) **do**
        **for** $i := 0$ **to** 1 **do**
30:         $proc := Winner[child][i]$;
31:         **if** $proc \neq \perp$ **then** $Enqueue(WaitingQueue, proc)$ **fi**
    **od od**;

32: $Winner$[Node($p$, MAX_LEVEL)][0] := $\perp$;                  /* reopen its leaf node */
33: $Remove(WaitingQueue, p)$;
34: $proc := Promoted$;
    **if** $(proc = p) \vee (proc = \perp)$ **then**
35:     $proc := Dequeue(WaitingQueue)$;
36:     $Promoted := proc$;
37:     **if** $proc \neq \perp$ **then** $Spin[proc] := true$ **fi**
    **fi**;

38: Signal()                                                  /* open the barrier */
**od**

**procedure** $AcquireNode$($lev$: 1..MAX_LEVEL)
    $n$ := Node($p$, $lev$);
39: $(prev, new) := fetch\text{-}and\text{-}update(Lock[n][0])$;
    $lock[lev] := new$;
    **if** $prev = \perp$ **then**
40:     $Winner[n][0] := p$;
        **return** PRIMARY_WINNER
    **else**
41:     $(prev, new) := fetch\text{-}and\text{-}update(WaiterLock[n])$;
        **if** $prev = \perp$ **then**
42:         $Waiter[n] := p$;
            **return** PRIMARY_WAITER
        **else**
43:         $(prev, new) := fetch\text{-}and\text{-}update(Lock[n][1])$;
            **if** $prev = \perp$ **then**
44:             $Winner[n][1] := p$;
                **return** SECONDARY_WINNER
            **else**
                **return** SECONDARY_WAITER
    **fi fi fi**
```

Figure 8.9: ALGORITHM T, continued.

Each node n is represented by the following variables: $\mathit{Lock}[n][0..1]$, $\mathit{Winner}[n][0..1]$, $\mathit{WaiterLock}[n]$, and $\mathit{Waiter}[n]$. Initially, all variables are $\perp$, representing an available node. Variables $\mathit{Lock}[n][0..1]$ and $\mathit{WaiterLock}[n]$ are used as lock variables, and are accessed by *fetch-and-update* and *fetch-and-reset* operations. If a process p invokes *fetch-and-update* on a lock variable while it has a value of $\perp$, then p "acquires" that variable. A process that acquires $\mathit{Lock}[n][0]$ (respectively, $\mathit{WaiterLock}[n]$, $\mathit{Lock}[n][1]$) becomes the primary winner (respectively, primary waiter, secondary winner), and stores its identity in $\mathit{Winner}[n][0]$ (respectively, $\mathit{Waiter}[n]$, $\mathit{Winner}[n][1]$).

At each node n (at level lev), process p tries to acquire some variable of that node by invoking *AcquireNode* (statements 3, 39–44 in Figure 8.9). If p becomes either the primary winner or the secondary winner, then it proceeds to the next level of the tree, as mentioned above. Otherwise, p stops at node n and waits until it is promoted (statement 4), as in ALGORITHM T0. If p becomes the primary winner, then it also stores the new value of $\mathit{Lock}[n][0]$ into a private variable $\mathit{lock}[\mathit{lev}]$ (where lev is the level of node n; statement 39 inside *AcquireNode*), to be used in its exit section.

Since the root node may have two winners, we now use a three-process mutual exclusion algorithm: process identifiers "0," "1," and "2" are used by the primary winner of the root node, the secondary winner of the root node, and a promoted process, respectively. Also, since each (non-root) node may have two winners, a double **for** loop is used to detect and enqueue the winners of child nodes (statements 30 and 31).

The following counterpart of Properties 3 and 4 holds in ALGORITHM T.

Property 5 If a process p acquires $\mathit{Lock}[n][0]$ at node $n > 1$, and if another process q later becomes the primary waiter at node n, then q examines every child of node n *after* $\mathit{Lock}[n][0]$ is released by p or by some other process on behalf of p.

Property 6 If a process p acquires $\mathit{Lock}[1][0]$, then p examines every child of node 1 (the root node) *after* $\mathit{Lock}[1][0]$ is released by p.

We now consider the behavior of a process p in its exit section, at a given node n. The behavior is slightly more complicated than that in ALGORITHM T0. (For brevity, we do not restate properties that are common to both ALGORITHM T0 and T.)

Case 1: p is a primary/secondary waiter at node n (statements 21–31). In this case, p is a promoted process. Hence, $p.\mathit{side}$ equals 2, and $p.\mathit{break\_level}$ equals the level of node n. (Recall that we use $p.v$ to represent p's private variable v; see page 51.) Therefore, statement 21 assigns $p.n := n$. If p is the primary waiter, then it also releases $WaiterLock[n]$ (statements 22–24). We now examine statements 25–29 by considering two cases.

- First, assume that n is a non-root node (*i.e.*, $n > 1$). In this case, p also checks if the primary winner still exists (statement 25), and if so, obtains the identity of the primary winner (statement 26), releases $Lock[n][0]$ (statements 27 and 28) and adds the primary winner to the waiting queue (statement 29). This is done in order to maintain Property 5, in the same way statements 15, 17, and 18 of ALGORITHM T0 maintain Property 3.

  Note that, unlike statement 15 of ALGORITHM T0, statement 26 cannot detect the primary winner (say, q) by simply executing "$proc := Winner[n][0]$." This is because q may have executed statement 39 and is about to execute statement 40. In such a case, $Winner[n][0]$ equals $\perp$ until q executes statement 40. To guard against such a case, statement 26 repeatedly reads $Winner[n][0]$ until $Winner[n][0] \neq \perp$ is true. (A simlar remark applies to statements 12 and 19, which are considered later.)

- Second, if n is the root node (*i.e.*, $n = 1$), then p skips statements 26–29 and leaves $Lock[n][0]$ and $Winner[n][0]$ unchanged, in the same way statements 15 and 16 of ALGORITHM T0 leave $Lock[1].winner$ unchanged.

Process p then examines every child of node n (specifically, $Lock[child][0..1]$, where $child$ is a child of n) to determine if any secondary waiters at node n exist (statements 30 and 31). p adds such processes to $WaitingQueue$.

Case 2: p is the primary winner at node n. In this case, p reads $Winner[n][0] = p$ at statement 9, and then executes statements 10–14.

Process p first resets $Winner[n][0]$ (statement 10), and then tries to release $Lock[n][0]$ by invoking *fetch-and-reset* (statement 11). If the old value of $Lock[n][0]$ (returned by *fetch-and-reset*) is different from $lock[lev]$, then there has been at least one other process, say q_1, that invoked *fetch-and-update* on $Lock[n][0]$ (by executing statement 39) and failed to acquire that variable. Therefore, q_1 must have tried (or is about to try) to acquire $WaiterLock[n]$ (by executing statement 41).

If q_1 succeeds in acquiring $WaiterLock[n]$, then it becomes the primary waiter; otherwise, there must be another primary waiter q_2. We consider two cases.

- First, assume that n is a non-root node (*i.e.*, $n > 1$). At some point of execution, there exists a primary waiter q (either q_1 or q_2). We claim that q cannot release $\mathit{WaiterLock}[n]$ before p executes statements 10–14. For the sake of contradiction, assume otherwise. Note that $\mathit{WaiterLock}[n]$ may be released only if q executes statements 22–24. Since statements 8–37 are protected by the barrier, it follows that q executes statements 22–29 before p executes statements 10–14. In this case, q reads $\mathit{Lock}[n][0] \neq \perp$ at statement 25, and then resets $\mathit{Winner}[n][0]$ and $\mathit{Lock}[n][0]$ at statements 27 and 28. Thus, p reads $\mathit{Winner}[n][0] \neq p$ at statement 9 (*i.e.*, p is no longer the primary winner), and hence p cannot execute statements 10–14, a contradiction.

  Therefore, eventually there exists a primary waiter q. Hence, p waits until $\mathit{Waiter}[n] \neq \perp$ is established (statement 12), at which point $\mathit{Waiter}[n] = q$ must hold. It then adds q to the waiting queue (statement 13).

- Second, assume that n is a root node (*i.e.*, $n = 1$). In this case, the primary waiter q may already have released $\mathit{WaiterLock}[1]$ by executing statement 24. (Note that q does not execute statements 26–29 in this case. Thus, p is still the primary winner.) Therefore, there is no guarantee that a primary waiter eventually exists. Thus, p skips statements 12 and 13.

  Note that, in this case, p later checks every child node of the root node (statements 30 and 31), thus ensuring that Property 6 holds. Since the primary waiter of the root node (if it exists) is the primary/secondary winner of some child node of the root node, p eventually detects the primary waiter, if it exists.

On the other hand, if the old value of $\mathit{Lock}[n][0]$ equals $\mathit{lock}[\mathit{lev}]$, then there are two possibilities: either **(i)** no other process accessed $\mathit{Lock}[n][0]$ after p acquired it, or **(ii)** at least *two* processes have done so. In either case, the *fetch-and-reset* operation has successfully released $\mathit{Lock}[n][0]$. As explained before, in Case (ii), the primary waiter of node n will be eventually detected by the secondary winner (if no other process elsewhere in the tree detects it). Thus, p does not execute statements 12 and 13.

Finally, p checks if the *fetch-and-reset* operation has established $\mathit{Lock}[n] = \perp$, and if not, establishes this condition by a simple write (statement 14). (Note that the *fetch-and-reset* operation is guaranteed to write $\perp$ only if $\mathit{Lock}[n][0]$ has the same value as written by p's last *fetch-and-update* operation (*i.e.*, $\mathit{Lock}[n][0] = \mathit{lock}[\mathit{lev}]$ holds), which may not be the case here.)

Case 3: $\boldsymbol{p}$ is the secondary winner at node $\boldsymbol{n}$. In this case, p reads $\mathit{Winner}[n][1] = p$ at statement 15, and then executes statements 16–20.

Process p first releases $\mathit{Winner}[n][1]$ and $\mathit{Lock}[n][1]$ by simple writes (statements 16 and 17). Then, p checks if there exists a primary waiter by examining $\mathit{WaiterLock}[n]$ (statement 18), and if so, adds the primary waiter to *WaitingQueue* (statements 19 and 20). □

In order to show that the algorithm is starvation-free, we only have to show that each primary or secondary waiter is eventually enqueued onto the global waiting queue.

First, consider a secondary waiter r. In order for r to become a secondary waiter, at the time when r invokes *fetch-and-update*($\mathit{WaiterLock}[n]$) (statement 41), there must be a primary waiter q of n. As shown below, q eventually executes its exit section, where it examines every child of n and adds r to the waiting queue.

Second, consider a primary waiter q at a node n. In order for q to become the primary waiter, at the time when q invokes *fetch-and-update*($\mathit{Lock}[n][0]$) (statement 39), there must be a primary winner p of n. We consider three cases.

- First, if p detects q in its exit section, then p clearly adds q to the waiting queue.

- Second, if p detects *another* primary waiter r, which enters and then exits its critical section before q acquires $\mathit{WaiterLock}[n]$, then r examines every child of n in its exit section. Since q must be a primary or secondary winner of some child of n, r discovers q and adds it to the waiting queue.

- Third, Assume that p does not detect the existence of the primary waiter in its exit section. That is, p finds $\mathit{prev} = \mathit{lock}[\mathit{lev}]$ at statement 11, and skips statements 12 and 13. In this case, there exists another process r that fails to acquire $\mathit{Lock}[n][0]$. If r becomes a primary waiter and then exits before q acquires $\mathit{WaiterLock}[n]$, then r detects q as in the second case. Thus, assume that r fails to acquire $\mathit{WaiterLock}[n]$. If r fails because some other process s has acquired $\mathit{WaiterLock}[n]$, then s has exited before q acquired $\mathit{WaiterLock}[n]$, and the reasoning is again similar to the second case. On the other hand, if r fails because q acquires $\mathit{WaiterLock}[n]$ before r, then either r eventually becomes the secondary winner, or r fails yet again because there exists another process s that is the secondary winner. In either case, there eventually exists a secondary winner (r or s) that detects q in its exit section.

Finally, note that every busy-waiting loop in Algorithm T is either a local-spin loop (statement 4) or is executed inside a mutually exclusive region (statements 12, 19, and 26). We can apply the technique in Section 8.2.2 and transform each of these non-local-spin loops into a local-spin loop (on DSM machines). A detailed correctness proof of Algorithm T is given in Appendix F. From the discussion so far, we have the following lemma.

Theorem 8.3 *Using any self-resettable fetch-and-ϕ primitive of rank $r \geq 3$, starvation-free mutual exclusion can be implemented with $\Theta(\log N/\log\log N)$ time complexity on either CC or DSM machines.* □

As in Section 8.2, we can combine Algorithm T with our adaptive algorithm (Algorithm A-LS) presented in Chapter 4 (see Theorem 8.2). Therefore, we have the following theorem.

Theorem 8.4 *Using any self-resettable fetch-and-ϕ primitive of rank $r \geq 3$, starvation-free adaptive mutual exclusion can be implemented with $O(\min(k, \log N/\log\log N))$ RMR time complexity on either CC or DSM machines, where k is point contention.* □

It can be shown that the $\Omega(\log N/\log\log N)$ lower bound presented in Chapter 5 applies to certain systems that use *fetch-and-ϕ* primitives of constant rank. The proof of that lower bound inductively extends computations so that information flow among processes is limited. If, at some induction step, a variable v is accessed by many processes, then information flow is kept low by ensuring that v may be assigned $O(1)$ different values during this induction step. Therefore, our lower bound applies to any *fetch-and-ϕ* primitive satisfying the following: *any consecutive invocations of the primitive by different processes can be ordered so that only $O(1)$ different values are returned.* It follows that, for self-resettable *fetch-and-ϕ* primitives with a constant rank of at least three that satisfy this condition, Algorithm T is asymptotically time-optimal. Examples of such primitives include a *fetch-and-increment/decrement* primitive with bounded range 0..2, a variant of *compare-and-swap* that allows two different compare values to be specified (2VCAS; see page 140), and the simultaneous execution of a *test-and-set* and a write operation on different bits of a variable.

8.4 Concluding Remarks

We have shown that any *fetch-and-ϕ* primitive of rank $r \geq 2$ can be used to implement a $\Theta(\log_{\min(r,N)} N)$ mutual exclusion algorithm, on either DSM or CC machines. $\Theta(\log_{\min(r,N)} N)$ is clearly optimal for $r = \Omega(N)$. For primitives of rank at least three that are self-resettable, we have presented a $\Theta(\log N/\log\log N)$ algorithm, which gives an asymptotic improvement in RMR time complexity for primitives of rank $o(\log N)$. This algorithm is time-optimal for certain self-resettable primitives of constant rank. In designing these algorithms, our main goal was to achieve certain asymptotic time complexities. In particular, we have not concerned ourselves with designing algorithms that can be practically applied. Indeed, it is difficult to design practical algorithms when assuming so little of the *fetch-and-ϕ* primitives being used. It is likely that by exploiting the semantics of a particular primitive, our algorithms could be optimized considerably.

As noted earlier in Section 1.4.4, we believe that the notion of rank defined in this chapter may be a suitable way of characterizing the "power" of primitives from the standpoint of blocking synchronization, much like the notion of a *consensus number*, which is used in Herlihy's wait-free hierarchy [41], reflects the "power" of primitives from the standpoint of nonblocking synchronization. Interestingly, primitives like *compare-and-swap* that are considered to be powerful according to Herlihy's hierarchy are weak from a blocking synchronization standpoint (since they are subject to the $\Omega(\log N/\log\log N)$ lower bound presented in Chapter 5). Also, primitives like *fetch-and-increment* and *fetch-and-store* that are considered to be powerful from a blocking synchronization standpoint are considered quite weak according to Herlihy's hierarchy. (They have consensus number two.) This difference arises because in nonblocking algorithms, the need to reach consensus is fundamental (as shown by Herlihy), while in blocking algorithms, the need to order competing processes is important.

The $\Theta(\log N/\log\log N)$ algorithm in Section 8.3 shows that $\Omega(\log N/\log\log N)$ is a tight lower bound for *some* class of synchronization primitives. Unfortunately, we have been unable to adapt the algorithm to work with only reads, writes, and comparison primitives. As stated in Chapter 5, we conjecture that $\Omega(\log N)$ is a tight lower bound for algorithms based on such operations.

CHAPTER 9

Conclusion

In this dissertation, we have presented many new results on shared-memory mutual exclusion algorithms. In this chapter, we summarize these results and discuss directions for future research.

9.1 Summary

In Chapter 3, we presented a mutual exclusion algorithm with $\Theta(N)$ space complexity and $\Theta(\log N)$ RMR (remote-memory-reference) time complexity on both CC and DSM systems. This algorithm was created by applying a series of simple transformations to Yang and Anderson's mutual exclusion algorithm (Algorithm YA-N) [84].

As described in Section 2.4, Attiya and Bortnikov presented an adaptive non-local-spin mutual exclusion algorithm under read/write atomicity [24]. Their algorithm achieves $O(n^2)$ space complexity, where n is an *a priori* upper bound on the number of concurrently active processes. Therefore, it is possible to construct an algorithm with space complexity independent of N, if one does not insist on local spinning. However, among local-spin algorithms, our algorithm is clearly optimal, because *every* process must have at least one spin variable.

This space-optimal algorithm is motivated by our work on adaptive mutual exclusion algorithms, presented in Chapter 4. In this chapter, we presented an adaptive local-spin algorithm for mutual exclusion under read/write atomicity. This is the first read/write algorithm that is adaptive under the RMR time complexity measure. (As mentioned earlier, Afek, Stupp, and Touitou [6] independently devised another such algorithm.) This algorithm (Algorithm A-LS) has $\Theta(N)$ space complexity, which is clearly optimal.

In Chapter 5, we established a lower bound of $\Omega(\log N/\log\log N)$ remote memory references for mutual exclusion algorithms based on reads, writes, or comparison primitives; for algorithms with comparison primitives, this bound only applies in non-LFCU systems. Our bound improves an earlier lower bound of $\Omega(\log\log N/\log\log\log N)$ established by Cypher. Given Algorithm YA-N [84], our bound is within a factor of $\Theta(\log\log N)$ of being optimal.

Given that constant-time algorithms based on *fetch-and-ϕ* primitives exist, this lower bound points to an unexpected weakness of *compare-and-swap*, which is widely regarded as being the most useful of all primitives to provide in hardware. In particular, this result implies that the *best* algorithm based on *compare-and-swap* can have RMR time complexity that is at most $\Theta(\log\log N)$ times smaller than that of the best algorithm based on reads and writes.

In Chapter 6, we established a lower bound that eliminates the possibility of an $o(k)$ adaptive mutual exclusion algorithm based on reads, writes, or comparison primitives, where k is either point or interval contention. One may ask whether the bounds presented in Chapters 5 and 6 together imply a lower bound of $\Omega(\min(k, \log N/\log\log N))$ RMRs for mutual exclusion algorithms (adaptive or not). Unfortunately, the answer is no. We have shown that $\Omega(k)$ time complexity is required *provided* N is sufficiently large. This leaves open the possibility that an algorithm might have $\Theta(k)$ time complexity for very "low" levels of contention, but $o(k)$ time complexity for "intermediate" levels of contention. Although our lower bound does not preclude such a possibility, we find it highly unlikely.

In Chapter 7, we presented a *nonatomic* local-spin mutual exclusion algorithm with $\Theta(\log N)$ worst-case RMR time complexity, which matches that of the best atomic algorithm (Algorithm YA-N) proposed to date. We also showed that for any N-process nonatomic algorithm, there exists a single-process execution in which the lone competing process accesses $\Omega(\log N/\log\log N)$ distinct variables in order to enter its critical section. This bound shows that fast and adaptive algorithms are impossible if variable accesses are nonatomic, even if caching techniques are used to avoid accessing the processors-to-memory interconnection network.

Finally, in Chapter 8, we defined the concept of a "rank" for *fetch-and-ϕ* primitives, and showed that any *fetch-and-ϕ* primitive of rank r can be used to implement a $\Theta(\log_{\min(r,N)} N)$ mutual exclusion algorithm, on either DSM or CC machines. $\Theta(\log_{\min(r,N)} N)$ is clearly optimal for $r = \Omega(N)$. For primitives of rank at least three that are self-resettable, we presented a $\Theta(\log N/\log\log N)$ algorithm, which gives an asymptotic improvement in RMR time complexity for primitives of rank $o(\log N)$. This algorithm is time-optimal for certain self-resettable primitives of constant rank. Either algorithm can be combined with the adaptive algorithm presented in Chapter 4 (Algorithm A-LS) to obtain a *fetch-and-ϕ*-based adaptive algorithm.

From the lower bound presented in Chapter 5 and the $\Theta(\log N/\log\log N)$ algorithm in Section 8.3, it follows that $\Omega(\log N/\log\log N)$ is a tight upper and lower bound for *some* class of synchronization primitives. As noted earlier on page 239, examples of such primitives include a *fetch-and-increment/decrement* primitive with bounded range 0..2, a variant of *compare-and-swap* that allows two different compare values to be specified, and the simultaneous execution of a *test-and-set* and a write operation on different bits of a variable.

9.2 Future Work

In this section, we discuss some of the remaining open problems and directions for further research on shared-memory mutual exclusion.

As explained earlier, the most efficient known local-spin algorithm based on reads and writes is Algorithm YA-N [84], which has $\Theta(\log N)$ time complexity. Thus, in establishing the $\Omega(\log N/\log\log N)$ lower bound presented in Chapter 5, we have *almost* succeeded in establishing the optimality of Algorithm YA-N. We conjecture that $\Omega(\log N)$ is a tight lower bound for the class of algorithms and systems to which our lower bound applies.

Conjecture: For any N-process mutual exclusion algorithm using reads, writes, and conditional primitives, there exists a history in which some process performs $\Omega(\log N)$ remote memory references to enter and exit its critical section.

It should be noted that Cypher's result guarantees that there exists no algorithm with *amortized* $\Theta(\log\log N/\log\log\log N)$ time complexity, while ours does not. This is because his bound is obtained by counting the total number of remote memory references in a computation, and by then dividing this number by the number of processes participating in that computation. In contrast, our result merely proves that there exists a computation H and a process p such that p executes $\Omega(\log N/\log\log N)$ critical events in H. Therefore, our result leaves open the possibility that the *average* number of remote memory references per process is less than $\Theta(\log N/\log\log N)$. We leave this issue for future research.

Open Problem: Can Cypher's amortized lower bound be improved?

Algorithm A-LS, presented in Chapter 4, has $O(\min(k, \log N))$ RMR time complexity. On the other hand, Attiya and Bortnikov [24] devised a *non-local-spin* adaptive algorithm with $O(\log k)$ system response time (see Section 2.4). It would be interesting to know whether the better features of these two algorithms could be combined. That is:

Open Problem: Can the mutual exclusion problem be solved with $O(k)$ RMR time complexity under the DSM model *and* with $O(\log k)$ system response time?

One may instead question if the RMR time complexity of Algorithm A-LS may be improved. Since the results of Chapter 6 preclude the possibility of an $o(k)$ algorithm, we conjecture that $\Omega(\min(k, \log N))$ is indeed a tight lower bound for adaptive mutual exclusion algorithms based on read, write, or comparison primitives (which would imply that Algorithm A-LS is optimal).

Conjecture: For any adaptive N-process mutual exclusion algorithm using reads, writes, and conditional primitives, and for any k ranging over $1 \leq k \leq N$, there exists a history in which some process experiences point contention k and generates $\Omega(\min(k, \log N))$ remote memory references to enter and exit its critical section.

One drawback of Algorithm A-LS is that every process starts execution by accessing the same variable. Hence, in practice, Algorithm A-LS may suffer from high hot-spot contention. Given the lower bounds of Anderson and Yang [22] (see Theorems 2.10 and 2.11), any fast or adaptive algorithm must use $\Omega(N^{\epsilon})$-writer variables. Thus, we know that write- and access-contention cannot be completely eliminated in designing adaptive algorithms. Although there has been work on formal complexity models that takes write/access-contention into account [35, 37], a time-complexity model that incorporates both memory locality and write/access-contention has not been devised so far. We leave this issue for future research.

The results of Chapter 7 suggest several avenues for further research. The most obvious is to close the gap between our $\Theta(\log N)$ algorithm and our $\Omega(\log N/\log\log N)$ lower bound. We conjecture that our lower bound can be improved to $\Theta(\log N)$.

Conjecture: For any N-process mutual exclusion algorithm using nonatomic reads and writes, there exists a single-process execution in which the lone competing process accesses $\Omega(\log N)$ distinct variables in order to enter and exit its critical section.

Another interesting question arises from the results of Chapter 7. Our proof hinges on the ability to "stall" nonatomic writes for arbitrarily long intervals. This gives rise to the following question: Is it possible to devise a nonatomic algorithm that is fast or adaptive if each write is guaranteed to complete within some bound Δ? We hope to resolve this question in future work.

The results of Chapter 8 only partially address the questions raised there regarding *fetch-and-ϕ*-based mutual exclusion algorithms. To remedy the situation, the definition of rank given in Chapter 8 needs to be justified. Towards this goal, we need to obtain a lower bound for mutual exclusion algorithms based on a generic *fetch-and-ϕ* primitive. However, assuming so little of the specific semantics of the given primitive, it seems extremely difficult to prove a nontrivial lower bound. We leave this question for future research.

In this dissertation, we studied algorithms based on atomic reads and writes (Chapters 3 and 4), nonatomic reads and writes (Chapter 7), and stronger synchronization primitives (Chapter 8). Most mutual exclusion algorithms presented in the literature fit within one of these categories, with one notable exception: timing-based algorithms. These algorithms are devised for semi-synchronous systems, in which the time required to execute a statement is upper-bounded. Timing-based algorithms exploit such bounds by allowing processes to delay their execution [7, 10, 56, 58].

In recent work [48], we presented several local-spin timing-based algorithms. Note that, for asynchronous systems (which are assumed throughout this dissertation), a lower bound of $\Omega(\log N/\log\log N)$ RMRs is fundamental, as established in Chapter 5. In [48], we showed that lower RMR time complexity is attainable in semi-synchronous systems with *delay* statements.

When assessing the time complexity of delay-based algorithms, the question of whether delays should be counted arises. We considered both possibilities. Also of relevance is whether delay durations are upper-bounded. Again, we considered both possibilities. For each of these possibilities, we presented an algorithm with either $\Theta(1)$ or $\Theta(\log\log N)$ time complexity, and established a matching time-complexity lower bound. It follows from these results that semi-synchronous systems allow mutual exclusion algorithms with substantially lower RMR time complexities than completely asynchronous systems, regardless of how one resolves the issues noted above.

The work mentioned above only considered the *known-delay* model [10, 56, 58], in which there is a known upper bound, denoted Δ, on the time required to access a shared variable. The known-delay model differs from the *unknown-delay* model [7], wherein the upper bound Δ is unknown *a priori*, and hence, cannot be used directly in an algorithm. Presently, we do not know if local-spin mutual exclusion with $o(\log N)$ RMR time complexity is possible for the unknown-delay model. Moreover, it is unknown whether RMR time complexity is a suitable means of comparing timing-based algorithms. We leave these questions for future research.

APPENDIX A

CORRECTNESS PROOF FOR ALGORITHM A-LS IN CHAPTER 4

In this appendix, we formally prove that ALGORITHM A-LS, presented in Chapter 4, satisfies the Exclusion property (at most one process executes its critical section at any time). In addition, we establish an invariant that implies that the algorithm is adaptive under the RMR measure. (The algorithm is easily seen to be starvation-free if the underlying algorithms used to implement the `ENTRY` and `EXIT` calls are starvation-free.) Our proof makes use of a number of auxiliary variables. In Figure A.1, ALGORITHM A-LS is shown with these added auxiliary variables. We have marked the lines of code that refer to auxiliary variables with a dash "—" to make them easier to distinguish.

We now define several terms that will be used in the proof. Unless stated otherwise, we assume that i and h range over $\{1, \ldots, T\}$, and that p and q range over $\{0, \ldots, N-1\}$.

Definition: We say that a process p is *a candidate to acquire splitter* i $(1 \leq i \leq T)$ if the condition $A(p, i)$, defined below, is true.

$$\begin{aligned} A(p,i) \;\equiv\; & p.node = i \;\wedge\; p@\{3..5, 8..14, 17..22, 24..41\} \;\wedge \\ & \big(p@\{17..19\} \;\Rightarrow\; X[i] = p\big) \;\wedge \\ & \big(p@\{17..22\} \;\Rightarrow\; \mathit{Reset}[i] = p.y\big) \;\wedge \\ & \big(p@\{3\} \;\Rightarrow\; p.dir = stop\big) \end{aligned}$$ □

By definition, if p is the winner of some round $\mathcal{R}(i, r)$ of splitter i, then p is also a candidate to acquire splitter i.

Definition: We define $i \xrightarrow{*} h$ to be true, where each of i and h $(1 \leq i, h \leq 2T + 1)$ is either a splitter or a "child" of a leaf splitter, if $i = h$ or if h is a descendent of i in the renaming tree. (When a process "falls off the end" of the renaming tree, it moves to level $L + 1$. The actual leaves of the renaming tree are at level L.) Formally, $\xrightarrow{*}$ is the transitive closure of the relation $\{(i, i),\ (i, 2i),\ (i, 2i + 1){:}\ 1 \leq i \leq T\}$. □

```
/* all constant, type, and variable declarations are as defined in Figure 4.7 except as noted here */
shared auxiliary variables
    Loc: array[0..2N − 1] of 0..2T + 1 initially 0;
    Dist: array[1..S] of (⊥, 0..S − 1);
    PC: array[0..N − 1, 1..T] of 0..2N initially 0

private auxiliary variable
    m: 1..T

initially (∀j : 1 ≤ j ≤ T :: Dist[j] = ⊥) ∧ (∀j : T < j ≤ S :: Dist[j] = j − T − 1)

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  node, level := 1, 0;
    — for m := 1 to T do PC[p, m] := 0 od;
    — UpdateLoc({p}, 1);
    /* descend renaming tree */
    repeat
2:      dir := AcquireNode(node);
3:      path[level] := (node, dir);
        if dir = left then
            level, node := level + 1, 2 · node
        elseif dir = right then
            level, node := level + 1, 2 · node + 1
        fi
    until (level > L) ∨ (dir = stop);
    if level ≤ L then       /* got a name */
        /* compete in the renaming tree,
                and then 2-process algorithm */
        for j := level downto 0 do
4:          Entry_3(path[j].node, path[j].dir)
        od;
5:      Entry_2(0)
    else     /* did not get a name */
        /* compete in the overflow tree,
                and then 2-process algorithm */
6:      Entry_N(p);
7:      Entry_2(1)
    fi;
8:  Critical Section;
    /* reset splitters */
    for j := min(level, L) downto 0 do
        n, dir := path[j].node, path[j].dir;
9:      ReleaseNode(n, dir)
    od;
    /* execute appropriate exit sections */
    if level ≤ L then
10:     Exit_2(0);
        for j := 0 to level do
11:         Exit_3(path[j].node, path[j].dir)
        od;
12:     ClearNode(node)
    else 13:Exit_2(1);
14:     Exit_N(p)
    fi;
    — Loc[p], Loc[p + N] := 0, 0
od
```

Figure A.1: ALGORITHM A-LS with auxiliary variables added. (Continued on the next page.)

```
function $AcquireNode(n\colon 1..T)\colon Dtype$
15: $X[n] := p$;
    — $\mathsf{UpdateLoc}(\{q\colon q@\{16..19\} \wedge$
    —                    $q.n = n\},\ 2n)$;
16: $y := Y[n]$;
    if $\neg y.free$ then
    — $\mathsf{UpdateLoc}(\{p\},\ 2n+1)$;
        return $right$
    fi;
17: $Y[n] := (false,\ 0)$;
18: $Inuse[p] := y.rnd$;
19: if $X[n] \neq p\ \vee$
20:        $Acquired[n]$ then
    — $\mathsf{UpdateLoc}(\{p\},\ 2n)$;
        return $left$
    fi;
21: $Round[y.rnd] := true$;
22: if $Reset[n] \neq y$ then
23:     $Round[y.rnd] := false$;
        return $left$
    fi;
24: $Acquired[n] := true$;
    return $stop$

procedure $ReleaseNode(n\colon 1..T,\ dir\colon Dtype)$
25: if $dir = right$ then return fi;
    — $\mathsf{Loc}[p] := n$;
26: $Y[n] := (false,\ 0)$;
27: $X[n] := p$;
    — $\mathsf{UpdateLoc}(\{q\colon q@\{16..19\} \wedge q.n = n\},\ 2n)$;
28: $y := Reset[n]$;
29: $Reset[n] := (false,\ y.rnd)$;
    — $\mathsf{UpdateLoc}(\{q\colon q@\{17..22\} \wedge q.n = n\},\ 2n)$;
30: if $(dir = stop\ \vee\ \neg Round[y.rnd])$ then
31:     $ptr := Check$;
32:     $usedrnd := Inuse[ptr]$;
33:     if $usedrnd \neq 0$ then
            $MoveToTail(Free,\ usedrnd)$
        fi;
34:     $Check := ptr + 1 \bmod N$;
35:     $Enqueue(Free,\ y.rnd)$;
36:     $nextrnd := Dequeue(Free)$;
37:     $Reset[n] := (true,\ nextrnd)$;
38:     $Y[n] := (true,\ nextrnd)$
    fi;
    if $dir = stop$ then
39:     $Round[y.rnd] := false$;
40:     $Inuse[p] := 0$
    fi

procedure $ClearNode(n\colon 1..T)$
41: $Acquired[n] := false$
```

Figure A.1: Algorithm A-LS with auxiliary variables added, continued.

Definition: We define $\mathsf{lev}(i)$ to be the level of splitter i in the renaming tree, *i.e.*, $\mathsf{lev}(i) = \lfloor \log i \rfloor$. □

We are now in a position to describe the auxiliary variables used in our proof. They are as follows.

- $\mathsf{Loc}[p]$ is the location of process p within the renaming tree, where $0 \leq p < 2N$. Note that $p.node \neq \mathsf{Loc}[p]$ is possible if p is *not* a candidate to acquire splitter $p.node$. (The exact value of $\mathsf{Loc}[p]$ is given in invariants (I48)–(I56), stated later.)

- $\mathsf{Dist}[r]$ specifies the distance of round number r from the head of *Free*. If r is not in *Free*, then $\mathsf{Dist}[r] = \perp$. $\mathsf{Dist}[r]$ is assumed to be updated within the procedures *Enqueue*, *Dequeue*, and *MoveToTail*.

One additional definition is needed before we consider the final set of auxiliary variables.

Definition: We define the *contention of splitter* i, denoted $C(i)$, as the number of processes p such that $\mathsf{Loc}[p]$ equals a splitter (or a child of a leaf splitter) in the subtree rooted at i. Formally,

$$C(i) \;\equiv\; \big|\{p : 0 \le p < 2N :: i \xrightarrow{*} \mathsf{Loc}[p]\}\big|. \qquad \square$$

The last set of auxiliary variables is as follows.

- $\mathsf{PC}[p, i]$ is the point contention experienced by process p in its entry section while at splitter i or one of its descendents in the renaming tree. In particular, when p moves down to splitter i, $\mathsf{PC}[p, i]$ is initialized to be $C(i)$, the contention of splitter i. From that point onward, $\mathsf{PC}[p, i]$ tracks the point contention of the subtree rooted at i as seen by process p, *i.e.*, the maximum value of $C(i)$ encountered since p moved down to spitter i. $\mathsf{PC}[p, i]$ is not used if process p is outside its entry section or if $i \xrightarrow{*} \mathsf{Loc}[p]$ is false.

The auxiliary procedure $\mathsf{UpdateLoc}(P, i)$ is called when a process enters its entry section (statement 1) and when a set P of processes moves to splitter i while in their entry sections (statements 15, 16, 20, 27, and 29). This procedure updates the value of $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p{+}N]$ to be i for each $p \in P$ (recall that both should have the same value while p is in its entry section), and sets the value of $\mathsf{PC}[p, i]$ as the current contention of the subtree rooted at i. It also updates the PC values of other processes to reflect the movement of processes in P.

```
     **procedure** UpdateLoc($P \subseteq \{0..N-1\}$, $i$: $1..T$)
u1:  **for all** $p \in P$ **do**   $\mathsf{Loc}[p],\ \mathsf{Loc}[p+N] := i,\ i$   **od**;
u2:  **for all** $p \in P$ **do**   $\mathsf{PC}[p, i] := C(i)$   **od**;
u3:  **for all** $q,\ h$ **s.t.** $q@\{2, 3, 15..24\} \wedge h \xrightarrow{*} \mathsf{Loc}[q]$ **do**
         **if** $\mathsf{PC}[q, h] < C(h)$ **then** $\mathsf{PC}[q, h] := C(h)$ **fi**
     **od**
```

Notice that the value of Loc array is changed directly at statements 14.p, 25.p, and 41.p (by our atomicity assumption, each of 14.p and 41.p establishes $p@\{0\}$ and includes the assignments to $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+n]$ that appear after statement 14.p). Because these statements are not within p's entry section, there is no need to update $\mathsf{PC}[p, i]$ (for any i). In addition, statement 25.p updates $\mathsf{Loc}[p]$ to move from a splitter to its ancestor, and statements 14.p and 41.p reinitialize $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ to 0 (*i.e.*, p is no longer within the renaming tree). Thus, contention does not increase within any subtree when these statements are executed.

A.1 List of Invariants

We will establish the Exclusion and Adaptivity properties by proving that the conjunction of a number of assertions is an invariant. This proves that each of these assertions individually is an invariant. These invariants are listed below.

(I) Invariants that give conditions that must hold if a process p is a candidate to acquire splitter i.

invariant $A(p,i) \ \wedge\ \big(p@\{3..8, 18..22, 24\} \ \vee\ (p@\{9, 25..38\} \ \wedge\ p.n = i)\big) \ \Rightarrow$
$\qquad Y[i] = (\mathit{false}, 0)$ (I1)

invariant $A(p,i) \ \wedge\ \big(p@\{3..8, 22, 24\} \ \vee\ (p@\{9, 25..38\} \ \wedge\ p.n = i)\big) \ \Rightarrow$
$\qquad \mathit{Round}[p.y.rnd] = \mathit{true}$ (I2)

invariant $A(p,i) \ \wedge\ \big(p@\{3..8, 24\} \ \vee\ (p@\{9, 25..37\} \ \wedge\ p.n = i)\big) \ \Rightarrow$
$\qquad \mathit{Reset}[i].rnd = p.y.rnd$ (I3)

invariant $\big(\exists p :: A(p,i) \ \wedge\ p@\{3..14, 25..41\}\big) = \big(\mathit{Acquired}[i] = \mathit{true}\big)$ (I4)

(II) Invariants that prevent "interference" of *Round* entries. These invariants are used to show that if $p@\{20..23\}$ holds and process p is *not* a candidate to acquire splitter $p.node$, then either $p.y.rnd$ is not in the *Free* queue, or it is "trapped" in the tail region of the queue. Therefore, there is no way $p.y.rnd$ can reach the head of *Free* and get assigned to another splitter.

invariant $p@\{17..19\} \ \wedge\ X[p.node] = p \ \Rightarrow$
$\qquad \mathit{Reset}[p.node] = p.y \ \wedge\ \mathsf{Dist}[p.y.rnd] = \bot \ \wedge$
$\qquad (\forall q :: q@\{37, 38\} \ \Rightarrow\ p.y.rnd \neq q.nextrnd)$ (I5)

invariant $p@\{20..23\} \ \wedge\ (\exists q :: q@\{34\}) \ \Rightarrow$
$\qquad \mathit{Reset}[p.node].rnd = p.y.rnd \ \vee$
$\qquad \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p) \ \mathbf{mod}\ N\big) - 2$ (I6)

invariant $p@\{20..23\} \ \wedge\ (\exists q :: q@\{35, 36\}) \ \Rightarrow$
$\qquad \mathit{Reset}[p.node].rnd = p.y.rnd \ \vee$
$\qquad \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 2$ (I7)

invariant $p@\{20..23\} \ \wedge\ \neg(\exists q :: q@\{34..36\}) \ \Rightarrow$
$\qquad \mathit{Reset}[p.node].rnd = p.y.rnd \ \vee$
$\qquad \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 3$ (I8)

invariant $p@\{20..23\} \ \wedge\ q@\{37, 38\} \ \Rightarrow\ p.y.rnd \neq q.nextrnd$ (I9)

invariant $p@\{23\} \ \wedge\ \mathit{Reset}[p.node].rnd = p.y.rnd \ \Rightarrow$

$$Reset[p.node].free = false \tag{I10}$$

invariant $Y[i].free = true \;\Rightarrow \mathsf{Dist}[Y[i].rnd] = \bot \;\wedge$
$$(\forall p :: p@\{37, 38\} \;\Rightarrow\; Y[i].rnd \neq p.nextrnd) \tag{I11}$$

invariant $p@\{37\} \;\Rightarrow\; (\forall i :: Reset[i].rnd \neq p.nextrnd)$ (I12)

invariant $p@\{38, 39\} \;\Rightarrow\; (\forall i :: Reset[i].rnd \neq p.y.rnd)$ (I13)

invariant $\mathsf{Dist}[Reset[i].rnd] = \bot \;\vee$
$(\exists p :: p@\{36\} \;\wedge\; p.n = i \;\wedge\; \mathsf{Dist}[Reset[i].rnd] = 2N) \;\vee$
$(\exists p :: p@\{37\} \;\wedge\; p.n = i \;\wedge\; \mathsf{Dist}[Reset[i].rnd] = 2N - 1)$ (I14)

invariant $i \neq h \;\Rightarrow\; Reset[i].rnd \neq Reset[h].rnd$ (I15)

invariant $p@\{20..23\} \;\wedge\; q@\{33\} \;\wedge\; (Check = p) \;\Rightarrow$
$$Reset[p.node].rnd = p.y.rnd \;\vee\; q.usedrnd = p.y.rnd \tag{I16}$$

(III) Invariants showing that certain regions of code are mutually exclusive.

invariant $A(p, i) \;\wedge\; A(q, i) \;\wedge\; p \neq q \;\Rightarrow$
$$\neg\big[p@\{17..20\} \;\wedge\; \big(q@\{3..8, 17..22, 24\} \;\vee\; (q@\{9, 25..38\} \;\wedge\; q.n = i)\big)\big] \tag{I17}$$

invariant $A(p, i) \;\wedge\; A(q, i) \;\wedge\; p \neq q \;\Rightarrow$
$$\neg(p@\{3..14, 21, 22, 24..41\} \;\wedge\; q@\{3..14, 21, 22, 24..41\}) \tag{I18}$$

invariant $A(p, i) \;\wedge\; q.n = i \;\Rightarrow\; \neg(p@\{17..19\} \;\wedge\; q@\{28..38\})$ (I19)

invariant $A(p, i) \;\wedge\; q.n = i \;\Rightarrow\; \neg(p@\{20, 21\} \;\wedge\; q@\{30..38\})$ (I20)

invariant $A(p, i) \;\wedge\; q.n = i \;\Rightarrow\; \neg(p@\{3..10, 22, 24\} \;\wedge\; q@\{31..38\})$ (I21)

invariant $A(p, i) \;\wedge\; \big(p@\{3..8, 20..22, 24\} \;\vee\; (p@\{9, 25..38\} \;\wedge\; p.n = i)\big) \;\wedge$
$q@\{20..23\} \;\wedge\; p \neq q \;\Rightarrow$
$$p.y.rnd \neq q.y.rnd \tag{I22}$$

(IV) Miscellaneous invariants that are either trivial or follow almost directly from the Exclusion property (I60). In (I37) and (I38), $\|Free\|$ denotes the length of the *Free* queue.

invariant $p@\{29\} \;\Rightarrow\; Reset[p.n] = p.y$ (I23)

invariant $p@\{30..37\} \;\Rightarrow\; Reset[p.n] = (false, p.y.rnd)$ (I24)

invariant $p@\{27..38\} \;\Rightarrow\; Y[p.n] = (false, 0)$ (I25)

invariant $Y[i].free = true \;\Rightarrow\; Y[i] = Reset[i]$ (I26)

invariant $Reset[i].rnd \neq 0$ (I27)

invariant $p@\{17..23\} \;\Rightarrow\; p.y.free = true$ (I28)

invariant $p@\{17..23, 29..40\} \;\Rightarrow\; p.y.rnd \neq 0$ (I29)

invariant $p@\{19..23\} \;\Rightarrow\; Inuse[p] = p.y.rnd$ (I30)

invariant $p@\{2, 3, 15..24\} \;\Rightarrow\; (0 \leq p.level \leq L) \;\wedge\; (1 \leq p.node \leq T)$ (I31)

invariant $p@\{9, 25..40\} \;\Rightarrow$
$1 \leq p.n \leq T \;\wedge\; p.n = p.path[p.j].node \;\wedge$
$p.dir = p.path[p.j].dir \;\wedge\; p.j = \mathsf{lev}(p.n)$ (I32)

invariant $p@\{9, 25..40\} \;\wedge\; p.dir = stop \;\Rightarrow\; p.n = p.node$ (I33)

invariant $p@\{9, 25..40\} \;\wedge\; p.n = p.node \;\Rightarrow\; p.dir = stop$ (I34)

invariant $p@\{2..41\} \;\Rightarrow\; (0 \leq p.level \leq L+1) \;\wedge\; (\mathsf{lev}(p.node) = p.level)$ (I35)

invariant $p@\{39, 40\} \;\Rightarrow\; p.n = p.node$ (I36)

invariant $\neg(\exists p :: p@\{36\}) \;\Rightarrow\; \|Free\| = 2N$ (I37)

invariant $(\exists p :: p@\{36\}) \;\Rightarrow\; \|Free\| = 2N + 1$ (I38)

invariant $p@\{32..34\} \;\Rightarrow\; p.ptr = Check$ (I39)

invariant $p@\{37, 38\} \;\Rightarrow\; \mathsf{Dist}[p.nextrnd] = \bot$ (I40)

invariant $p@\{36\} \;\Rightarrow\; \mathsf{Dist}[p.y.rnd] = 2N$ (I41)

invariant $p@\{37\} \;\Rightarrow\; \mathsf{Dist}[p.y.rnd] = 2N - 1$ (I42)

invariant $(1 \leq i \leq S) \;\wedge\; Round[i] = true \;\Rightarrow$
$\big(\exists p :: p.y.rnd = i \;\wedge$
$\big(p@\{3..5, 8, 22..24\} \;\vee\; (p@\{9, 25..39\} \;\wedge\; p.n = p.node)\big) \;\wedge$
$\big(p@\{3..5, 8\} \;\Rightarrow\; (p.dir = stop \;\wedge\; 1 \leq p.node \leq T)\big)\big)$ (I43)

invariant $p@\{4..14, 25..41\} \;\wedge\; (p.node \leq T) \;\Rightarrow\; p.path[p.level] = (p.node, stop)$ (I44)

invariant $p@\{2..41\} \;\wedge\; (2i \xrightarrow{*} p.node) \;\Rightarrow\; p.path[\mathsf{lev}(i)] = (i, left)$ (I45)

invariant $p@\{3\} \;\wedge\; (p.level > 0) \;\Rightarrow$
$p.path[0].node = 1 \;\wedge$
$(\forall l : 0 \leq l < p.level - 1 :: p.path[l].node \xrightarrow{*} p.path[l+1].node) \;\wedge$
$p.path[p.level - 1].node \xrightarrow{*} p.node$ (I46)

invariant $p@\{26..38\} \;\Rightarrow\; (p.n = p.node) \;\vee\; (2 \cdot p.n \xrightarrow{*} p.node)$ (I47)

(V) Invariants that give the value of the auxiliary variable Loc.

invariant $p@\{0, 1\} \;\Rightarrow\; \mathsf{Loc}[p] = 0 \;\wedge\; \mathsf{Loc}[p+N] = 0$ (I48)

invariant $p@\{2, 15\} \;\Rightarrow\; \mathsf{Loc}[p] = p.node \;\wedge\; \mathsf{Loc}[p+N] = p.node$ (I49)

invariant $p@\{16\} \;\Rightarrow$
$\big(X[p.node] = p \;\Rightarrow\; \mathsf{Loc}[p] = p.node \;\wedge\; \mathsf{Loc}[p+N] = p.node\big) \;\wedge$
$\big(X[p.node] \neq p \;\Rightarrow\; \mathsf{Loc}[p] = 2 \cdot p.node \;\wedge\; \mathsf{Loc}[p+N] = 2 \cdot p.node\big)$ (I50)

invariant $p@\{17..24\} \;\Rightarrow$

$$\big(A(p, p.node) \;\Rightarrow\; \mathsf{Loc}[p] = p.node \;\wedge\; \mathsf{Loc}[p+N] = p.node\big) \;\wedge$$
$$\big(\neg A(p, p.node) \;\Rightarrow\; \mathsf{Loc}[p] = 2\cdot p.node \;\wedge\; \mathsf{Loc}[p+N] = 2\cdot p.node\big) \qquad \text{(I51)}$$

invariant $p@\{3\} \;\Rightarrow$
$$\big(p.dir = stop \;\Rightarrow\; \mathsf{Loc}[p] = p.node \;\wedge\; \mathsf{Loc}[p+N] = p.node\big) \;\wedge$$
$$\big(p.dir = left \;\Rightarrow\; \mathsf{Loc}[p] = 2\cdot p.node \;\wedge\; \mathsf{Loc}[p+N] = 2\cdot p.node\big) \;\wedge$$
$$\big(p.dir = right \;\Rightarrow \mathsf{Loc}[p] = 2\cdot p.node + 1 \;\wedge$$
$$\mathsf{Loc}[p+N] = 2\cdot p.node + 1\big) \qquad \text{(I52)}$$

invariant $p@\{4..14, 25..41\} \;\Rightarrow\; \mathsf{Loc}[p+N] = p.node$ (I53)

invariant $p@\{4..8\} \;\Rightarrow\; \mathsf{Loc}[p] = p.node$ (I54)

invariant $p@\{9, 25\} \;\Rightarrow\; p.n \xrightarrow{*} \mathsf{Loc}[p] \xrightarrow{*} p.node$ (I55)

invariant $p@\{26..40\} \;\Rightarrow\; \mathsf{Loc}[p] = p.n$ (I56)

(VI) Invariants that limit the maximum number of processes allowed within a given subtree of the renaming tree.

invariant $Y[i].free = false \;\Rightarrow$
$$\big(\exists p :: \big(p@\{3..8, 18..24\} \;\vee\; (p@\{9, 25..38\} \;\wedge\; p.n = i)\big) \;\wedge$$
$$(p.node = i) \;\wedge\; (p@\{3\} \;\Rightarrow\; p.dir \neq right)\big) \;\vee$$
$$\big(\exists p :: p@\{2..9, 15..40\} \;\wedge\; (2i \xrightarrow{*} p.node) \;\wedge$$
$$(p@\{9, 25..40\} \;\Rightarrow\; 2i \xrightarrow{*} p.n)\big) \;\vee$$
$$\big(\exists p :: p@\{9, 25..38\} \;\wedge\; p.n = i \;\wedge\; (p@\{9, 25\} \;\Rightarrow\; 2i \xrightarrow{*} \mathsf{Loc}[p])\big) \qquad \text{(I57)}$$

invariant $p@\{2, 3, 15..24\} \;\wedge\; (i \xrightarrow{*} \mathsf{Loc}[p]) \;\Rightarrow\; \mathsf{PC}[p, i] \geq C(i)$ (I58)

invariant $p@\{2, 3, 15..24\} \;\wedge\; (i \xrightarrow{*} \mathsf{Loc}[p]) \;\wedge\; (2 \leq i \leq T) \;\Rightarrow$
$$\mathsf{PC}[p, i] < \mathsf{PC}[p, \lfloor i/2 \rfloor] \qquad \text{(I59)}$$

(VII) Invariants that prove the Exclusion property and adaptivity under the RMR measure.

invariant (Exclusion) $\big|\{p :: p@\{8..10, 13, 25..40\}\}\big| \leq 1$ (I60)

invariant (Adaptivity) $p@\{4..8\} \;\Rightarrow\; \mathsf{lev}(p.node) < \mathsf{PC}[p, 1]$ (I61)

We now prove that each of (I1)–(I61) is an invariant. For each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.) If I is an implication (which is the case for most of our invariants), then it suffices to check only those program statements that may establish the antecedent of I, or that may falsify the consequent if executed while the antecedent holds. The following lemma is used in several of the proofs.

Lemma A.1 If $1 \leq i \leq T$ holds and t and u are consecutive states such that $A(p, i)$ is false at t but true at u, and if all the invariants stated above hold at t, then the following are true.

- u is reached from t via the execution of statement $16.p$.
- $p@\{17\}$ is established at u.
- $p.node = i \ \wedge \ Y[i].free = true$ holds at both t and u.

Proof: The only statements that could potentially establish $A(p, i)$ are the following.

- $1.p$ and $3.p$, which could establish $p.node = i$.
- $7.p$, $16.p$, and $23.p$, which establish $p@\{3..5, 8..14, 17..22, 24..41\}$.
- $15.p$ and $27.p$, which could establish $X[i] = p$.
- $19.p$, which falsifies $p@\{17..19\}$.
- $16.p$, $28.p$, $29.q$, and $37.q$, where q is any arbitrary process, which could establish $Reset[i] = p.y$.
- $19.p$, $20.p$, and $22.p$, which could falsify $p@\{17..22\}$.
- $24.p$, which could establish $p@\{3\} \ \wedge \ p.dir = stop$.

We now show that none of these statements other than $16.p$ can establish $A(p, i)$, and $16.p$ can do so only if the conditions specified in the lemma are met.

Statements $1.p$ and $15.p$ establish $p@\{2, 16\}$. Therefore, they cannot establish $A(p, i)$.

Statement $3.p$ establishes $p@\{2, 4, 6\}$. If it establishes $p@\{2, 6\}$, then $A(p, i)$ is false at u. If it establishes $p@\{4\}$, then $p.dir = stop$ holds at t. In this case, $3.p$ can establish $A(p, i)$ only if $p.node = i$ also holds at t. But this implies that $A(p, i)$ holds at t, which is a contradiction.

Statement $7.p$ can establish $A(p, i)$ by establishing $p@\{8\}$ only if $p.node = i$ holds at t. However, note that statement $7.p$ can be executed only if $p.level > L$ holds. By (I35), this implies $p.node > T$, a contradiction.

Statement $16.p$ can establish $A(p, i)$ by establishing $p@\{17\}$ only if $p.node = i \ \wedge$ $Y[i].free = true$ holds at t. But then this condition also holds at u.

Statement 19.p establishes either $p@\{3\}$ or $p@\{20\}$. If it establishes $p@\{3\}$, then it also establishes $p.dir = \mathit{left}$, in which case $A(p, i)$ is false at u. If 19.p establishes $p@\{20\}$, then $A(p, i)$ holds at u only if $p.node = i \ \wedge\ X[i] = p \ \wedge\ \mathit{Reset}[i] = p.y$ holds at t. But this contradicts our assumption that $A(p, i)$ is false at t.

Statements 20.p and 23.p can establish $A(p, i)$ only if they establish $p@\{3\}$. In this case, they also establish $p.dir = \mathit{left}$, which implies that $A(p, i)$ is false.

Statement 22.p establishes either $p@\{23\}$ or $p@\{24\}$. If it establishes $p@\{23\}$, then $A(p, i)$ is false at u. On the other hand, if it establishes $p@\{24\}$, then $A(p, i)$ holds at u only if $p.node = i \ \wedge\ \mathit{Reset}[i] = p.y$ holds at t. But this contradicts our assumption that $A(p, i)$ is false at t.

Statements 24.p, 27.p, 28.p, 29.p, and 37.p can establish $A(p, i)$ only if $p.node = i$ holds at both t and u. But then $A(p, i)$ holds at t, a contradiction.

Statement 29.q, where $q \neq p$, can establish $A(p, i)$ only by establishing $\mathit{Reset}[i] = p.y$ when $p@\{17..22\} \ \wedge\ q.n = i$ holds. In this case, 29.q establishes $\mathit{Reset}[i].\mathit{free} = \mathit{false}$. By (I28), if $p@\{17..22\}$ holds at t, then $p.y.\mathit{free} = \mathit{true}$ holds at t, and hence also at u. Therefore, 29.q cannot establish $\mathit{Reset}[i] = p.y$.

Statement 37.q can establish $A(p, i)$ only by establishing $\mathit{Reset}[i] = p.y$ when the expression $p@\{17..22\} \ \wedge\ p.node = i \ \wedge\ q@\{37\} \ \wedge\ q.n = i$ holds at t. We consider the two cases $p@\{17..19\}$ and $p@\{20..22\}$ separately.

Suppose that $p@\{17..19\}$ holds at t. In this case, 37.q can establish $A(p, i)$ only if $X[i] = p$ holds at t. By (I5), this implies that $\mathit{Reset}[i] = p.y$ holds as well. Thus, we have $p@\{17..19\} \ \wedge\ p.node = i \ \wedge\ X[i] = p \ \wedge\ \mathit{Reset}[i] = p.y$ at state t, which implies that $A(p, i)$ is true at t, a contradiction.

Finally, suppose that $p@\{20..22\}$ holds at t. In this case, by (I9), $p.y.\mathit{rnd} \neq q.\mathit{nextrnd}$ also holds. Therefore, $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ holds after the execution of 37.q, which implies that $A(p, i)$ is false. □

A.2 Proof of the Exclusion Property

We begin by proving those invariants needed to establish the Exclusion property (I60).

$$\textbf{invariant}\ \ A(p, i) \ \wedge\ \big(p@\{3..8, 18..22, 24\} \ \vee\ (p@\{9, 25..38\} \ \wedge\ p.n = i)\big) \ \Rightarrow \\ Y[i] = (\mathit{false}, 0) \qquad \text{(I1)}$$

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is 16.p. However, if statement 16.p establishes $A(p, i)$, then it also establishes $p@\{17\}$. Hence, it cannot falsify (I1).

The only statements that can establish $p@\{3..8, 18..22, 24\}$ are 16.p, 17.p, and 23.p. Statement 17.p establishes the consequent. Statements 16.p and 23.p can establish $p@\{18..22, 3..8\}$ only by establishing $p@\{3\}$, in which case they also establish $p.dir \neq stop$. Thus, if these statements establish $p@\{18..22, 3..8\}$, then they also falsify $A(p, i)$.

The only statement that can establish $p@\{9, 25..38\} \ \wedge\ p.n = i$ while $A(p, i)$ holds is 8.p. (Note that $A(p, i)$ implies $p.node = i$ by definition. Thus, if statement 40.p establishes $p@\{9\}$, then $p.n \neq i$ holds after its execution.) In this case, the antecedent holds before the execution of 8.p. Thus, although statement 8.p may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statement 38.q, where q is any arbitrary process. If $q = p$, then statement 38.q establishes $(p@\{9\} \ \wedge\ p.n \neq i) \ \vee\ p@\{10, 13, 39\}$, which implies that the antecedent is false. Suppose that $q \neq p$. Statement 38.q can falsify the consequent only if executed when $q.n = i$ holds. However, by (I19), (I20), and (I21), the antecedent and $q@\{38\} \ \wedge\ q.n = i$ cannot hold simultaneously. $\Box$

invariant $A(p, i) \ \wedge\ \big(p@\{3..8, 22, 24\} \ \vee\ (p@\{9, 25..38\} \ \wedge\ p.n = i)\big) \ \Rightarrow$
$$Round[p.y.rnd] = true \qquad \text{(I2)}$$

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is 16.p. However, if statement 16.p establishes $A(p, i)$, then it also establishes $p@\{17\}$. Hence, it cannot falsify (I2).

The condition $p@\{3..8, 22, 24\}$ may be established only by statements 16.p, 19.p, 20.p, 21.p, and 23.p. Statement 21.p establishes the consequent. If statements 16.p, 19.p, 20.p, and 23.p establish $p@\{3..8, 22, 24\}$, then they also establish $p@\{3\} \wedge p.dir \neq stop$, which implies that $A(p, i)$ is false after the execution of each of these statements.

The only statement that can establish $p@\{9, 25..38\} \ \wedge\ p.n = i$ while $A(p, i)$ holds is 8.p. (Note that $A(p, i)$ implies $p.node = i$ by definition.) In this case, the antecedent holds before the execution of 8.p. Thus, although statement 8.p may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statements 16.p and 28.p (which may change the value of $p.y.rnd$) and 23.q and 39.q (which assign the value *false* to an element of the *Round* array), where q is any arbitrary process. Statement 16.p establishes

$p@\{17\} \ \vee \ (p@\{3\} \ \wedge \ p.dir = right)$, which implies that the antecedent is false. If both the antecedent and $p@\{28\}$ hold, then by (I3), $Reset[i].rnd = p.y.rnd$ holds. It follows that statement $28.p$ cannot change the value of $p.y.rnd$ when the antecedent is true, and hence cannot falsify (I2).

If $q = p$, then statement $23.q$ establishes $p@\{3\} \ \wedge \ p.dir = left$ and statement $39.q$ establishes $p@\{40\}$, both of which imply that the antecedent is false.

Suppose that $q \neq p$. In this case, statements $23.q$ and $39.q$ may falsify the consequent only if $p.y.rnd = q.y.rnd$ holds. By (I22), $p.y.rnd = q.y.rnd \ \wedge \ q@\{23\}$ implies that the antecedent of (I2) is false. Thus, statement $23.q$ cannot falsify (I2).

By (I60), if the antecedent and $q@\{39\}$ hold, then we have $A(p, i) \wedge p@\{3..7, 22, 24\}$. By the definition of $A(p, i)$, $A(p, i) \ \wedge \ p@\{22\}$ implies that $Reset[i].rnd = p.y.rnd$ holds. By (I3), $A(p, i) \ \wedge \ p@\{3..7, 24\}$ also implies that $Reset[i].rnd = p.y.rnd$ holds. By (I13), $Reset[i].rnd = p.y.rnd \ \wedge \ q@\{39\}$ implies that $p.y.rnd \neq q.y.rnd$. Thus, statement $39.q$ cannot falsify (I2). □

invariant $A(p, i) \ \wedge \ \big(p@\{3..8, 24\} \ \vee \ (p@\{9, 25..37\} \ \wedge \ p.n = i)\big) \ \Rightarrow$
$$Reset[i].rnd = p.y.rnd \qquad \text{(I3)}$$

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is $16.p$. However, if statement $16.p$ establishes $A(p, i)$, then it also establishes $p@\{17\}$. Hence, it cannot falsify (I3).

The condition $p@\{3..8, 24\}$ may be established only by $16.p$, $19.p$, $20.p$, $22.p$, and $23.p$. If statements $16.p$, $19.p$, $20.p$, and $23.p$ establish $p@\{3..8, 24\}$, then they also establish $p@\{3\} \wedge p.dir \neq stop$, which implies that $A(p, i)$ is false after the execution of each of these statements. Statement $22.p$ can establish the antecedent only if $Reset[i] = p.y$ holds. Hence, it preserves (I3).

The only statement that can establish $p@\{9, 25..37\} \ \wedge \ p.n = i$ while $A(p, i)$ holds is $8.p$. (Note that $A(p, i)$ implies $p.node = i$ by definition.) In this case, the antecedent holds before the execution of $8.p$. Thus, although statement $8.p$ may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statements $16.p$ and $28.p$ (which update $p.y.rnd$) and $29.q$ and $37.q$ (which update $Reset[i].rnd$), where q is any arbitrary process. The antecedent is false after the execution of $16.p$ (as explained above) and $37.p$ (which establishes $p@\{38\}$). If statement $28.p$ or $29.p$ is executed while the antecedent holds, then the consequent is preserved.

This leaves only statements $29.q$ and $37.q$, where $q \neq p$. By (I23), $29.q$ cannot change $\mathit{Reset}[i].\mathit{rnd}$, and hence, cannot falsify (I3). Statement $37.q$ may falsify the consequent only if $q.n = i$ holds. By (I21), $q.n = i \;\wedge\; q@\{37\}$ implies that the antecedent of (I3) is false. Thus, statement $37.q$ cannot falsify (I3). $\Box$

invariant $\big(\exists p :: A(p, i) \;\wedge\; p@\{3..14, 25..41\}\big) = \big(\mathit{Acquired}[i] = \mathit{true}\big)$ (I4)

Proof: By the definition of $A(p, i)$, the left-hand side of (I4) is equivalent to $\big(\exists p :: p.\mathit{node} = i \;\wedge\; p@\{3..5, 8..14, 25..41\} \;\wedge\; (p@\{3\} \;\Rightarrow\; p.\mathit{dir} = \mathit{stop})\big)$. Thus, the left-hand side may be established or falsified only by statements $1.p$ (which may update $p.\mathit{node}$), $3.p$ (which may update $p.\mathit{node}$ and also falsify $p@\{3\}$), $16.p$, $19.p$, $20.p$, $23.p$, and $24.p$ (which may establish $p@\{3\}$ and also update $p.\mathit{dir}$), and $7.p$, $14.p$, and $41.p$ (which may establish or falsify $p@\{3..5, 8..14, 25..41\}$). $24.p$ and $41.p$ are also the only statements that may establish or falsify the right-hand side of (I4) (p can be any process here).

The left-hand side of (I4) is false before and after the execution of each of $1.p$, $16.p$, $19.p$, $20.p$, and $23.p$. (Note that, if one of $16.p$, $19.p$, $20.p$, or $23.p$ establishes $p@\{3\}$, then it also establishes $p.\mathit{dir} \neq \mathit{stop}$.) If statement $3.p$ is executed when $p.\mathit{dir} = \mathit{stop}$ holds, then it establishes $p@\{4\}$, and does not update $p.\mathit{node}$. Thus, the left-hand side is true before and after its execution. On the other hand, if statement $3.p$ is executed when $p.\mathit{dir} \neq \mathit{stop}$ holds, then it establishes $p@\{2, 6\}$, and hence the left-hand side is false before and after its execution. It follows that statement $3.p$ cannot establish or falsify the left-hand side.

Statement $24.p$ establishes the left-hand side if and only if it also establishes the right-hand side. Statement $41.p$ falsifies the right-hand side if and only if executed when $p.\mathit{node} = i$ holds, in which case $A(p, i)$ holds by definition. By (I18), this implies that the left-hand side of (I4) is falsified.

Statements $7.p$ and $14.p$ may be executed only when $p.\mathit{level} > L$. By (I35), $p.\mathit{level} > L$ implies that $p.\mathit{node} > T$. Because $i \leq T$ (by assumption), this implies that $A(p, i)$ is false both before and after any of these statements is executed. Thus, these statements can neither establish nor falsify the left-hand side of (I4). $\Box$

invariant $p@\{17..19\} \;\wedge\; X[p.\mathit{node}] = p \;\Rightarrow$

$$\begin{aligned}&\mathit{Reset}[p.\mathit{node}] = p.y \;\wedge\; \mathsf{Dist}[p.y.\mathit{rnd}] = \bot \;\wedge\\ &(\forall q :: q@\{37, 38\} \;\Rightarrow\; p.y.\mathit{rnd} \neq q.\mathit{nextrnd})\end{aligned} \qquad \text{(I5)}$$

Proof: The antecedent may be established only by statements 16.p (which establishes $p@\{17..19\}$), 1.p and 3.p (which update $p.node$), and 15.p and 27.p (which may establish $X[p.node] = p$). However, statements 1.p, 3.p, 15.p, and 27.p establish $p@\{2, 4, 6, 16, 28\}$ and hence cannot establish the antecedent. Also, by (I26) and (I11), if 16.p establishes the antecedent, then it also establishes the consequent.

The consequent may be falsified only by statements 16.p and 28.p (which may change the value of $p.y$), 1.p and 3.p (which may update $p.node$), 36.r (which may establish $r@\{37, 38\} \wedge p.y.rnd = r.nextrnd$), 29.$r$ and 37.r (which may update $Reset[p.node]$), and 35.r (which may falsify $\mathsf{Dist}[p.y.rnd] = \bot$), where r is any arbitrary process. However, 16.p preserves (I5) as shown above. Furthermore, the antecedent is false after the execution of 1.p, 3.p, and 28.p and also after the execution of each of 29.r, 35.r, 36.r, and 37.r if $r = p$.

Consider statements 29.r, 35.r, and 37.r, where $r \neq p$. If the antecedent and consequent of (I5) both hold, then by (I31), $p.node \leq T$ holds, and hence $A(p, p.node)$ holds. Statements 29.r and 37.r may falsify the consequent only if $r.n = p.node$ holds. Similarly, statement 35.r may falsify the consequent only if $r.y.rnd = p.y.rnd$. If $r@\{35\} \wedge r.y.rnd = p.y.rnd$ and the consequent both hold, then by (I24), we have $Reset[r.n].rnd = Reset[p.node].rnd$. By (I15), this implies that $r.n = p.node$. Therefore, each of these statements may falsify the consequent only if $r.n = p.node$ holds. However, by (I19), $r.n = p.node \wedge r@\{29, 35, 37\} \wedge A(p, p.node)$ implies that $p@\{17..19\}$ is false. Thus, these statements cannot falsify (I5).

Finally, statement 36.r may establish $p.y.rnd = r.nextrnd$ only if $p.y.rnd$ is at the head of *Free* queue, *i.e.*, $\mathsf{Dist}[p.y.rnd] = 0$. But this implies that $\mathsf{Dist}[p.y.rnd] = \bot$ is false. Because (I5) is assumed to hold prior to the execution of 36.r, this implies that the antecedent of (I5) is false before 36.r is executed. Thus, the antecedent is also false after the execution of 36.r. □

invariant $p@\{20..23\} \wedge (\exists q :: q@\{34\}) \Rightarrow$

$$Reset[p.node].rnd = p.y.rnd \vee$$
$$\mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p) \textbf{ mod } N\big) - 2 \qquad \text{(I6)}$$

Proof: The antecedent may be established only by statements 19.p and 33.q, where q is any process. Statement 19.p may establish the antecedent only if executed when $X[p.node] = p$ holds, in which case $Reset[p.node].rnd = p.y.rnd$ holds, by (I5).

Statement 33.q may establish the consequent only if executed when $p@\{20..23\}$ holds. By (I60), $q@\{33\} \ \wedge \ p@\{20..23\}$ implies that the antecedent of (I8) holds. By the consequent of (I8),

$$\big(\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}\big) \vee \big(\mathsf{Dist}[p.y.\mathit{rnd}] > 2N-2\cdot\big((\mathit{Check}-p-1) \ \mathbf{mod} \ N\big)-3\big) \tag{A.1}$$

holds as well. Now, consider the following three cases.

- $\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ holds before 33.q is executed. In this case, after the execution of 33.q, $\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ continues to hold, so (I6) is preserved.

- $\mathit{Check} = p \ \wedge \ \mathit{Reset}[p.\mathit{node}].\mathit{rnd} \neq p.y.\mathit{rnd}$ holds before 33.q is executed. In this case, by (I29), $p.y.\mathit{rnd} \neq 0$ holds, and by (I16), $q.\mathit{usedrnd} = p.y.\mathit{rnd}$ holds. Therefore, procedure *MoveToTail* is called. By (I60) and (I37), $q@\{33\}$ implies that $\|\mathit{Free}\| = 2N$. Thus, statement 33.q establishes $\mathsf{Dist}[p.y.\mathit{rnd}] = 2N - 1$, which implies the consequent.

- $\mathit{Check} \neq p \wedge \mathit{Reset}[p.\mathit{node}].\mathit{rnd} \neq p.y.\mathit{rnd}$ holds before 33.q is executed. By (A.1), this implies $\mathsf{Dist}[p.y.\mathit{rnd}] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod} \ N\big) - 3$. Note that the value of $\mathsf{Dist}[p.y.\mathit{rnd}]$ can decrease by at most one by a call to *MoveToTail*. Note also that if $\mathit{Check} \neq p$, then $(\mathit{Check} - p) \ \mathbf{mod} \ N = \big((\mathit{Check} - p - 1) \ \mathbf{mod} \ N\big) + 1$. Therefore, after the execution of 33.q,

$$\begin{aligned}\mathsf{Dist}[p.y.\mathit{rnd}] &> \big[2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod} \ N\big) - 3\big] - 1 \\ &= 2N - 2 \cdot \big(((\mathit{Check} - p) \ \mathbf{mod} \ N) - 1\big) - 4 \\ &= 2N - 2 \cdot \big((\mathit{Check} - p) \ \mathbf{mod} \ N\big) - 2.\end{aligned}$$

The consequent may be falsified only by statements 1.p and 3.p (which may update $p.\mathit{node}$), 16.p and 28.p (which may change the value of $p.y$), 29.r and 37.r (which may update $\mathit{Reset}[p.\mathit{node}].\mathit{rnd}$), 33.$r$, 35.$r$, and 36.$r$ (which may update $\mathsf{Dist}[p.y.\mathit{rnd}]$), and 34.$r$ (which may change the value of *Check*), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements 1.p, 3.p, 16.p, and 28.p. By (I60), statements 29.r, 33.r, 35.r, 36.r, and 37.r cannot be executed while the antecedent holds. Finally, by (I60), statement 34.r falsifies the antecedent. □

invariant $p@\{20..23\} \ \wedge\ (\exists q :: q@\{35, 36\}) \ \Rightarrow$

$$\begin{aligned}&\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd} \ \vee \\ &\mathsf{Dist}[p.y.\mathit{rnd}] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 2\end{aligned} \tag{I7}$$

Proof: The antecedent may be established only by statements 19.p and 34.q, where q is any process. Statement 19.p may establish the antecedent only if executed when $X[p.\mathit{node}] = p$ holds, in which case $\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ holds, by (I5). Statement 34.q can establish the antecedent only if $q@\{34\} \wedge p@\{20..23\}$ holds. This implies that the consequent of (I6) holds before statement 34.q is executed. By (I39), statement 34.q increments the value of *Check* by 1 modulo-N. Thus, the consequent of (I7) is established.

The consequent may be falsified only by statements 1.p and 3.p (which may update $p.\mathit{node}$), 16.p and 28.p (which may change the value of $p.y$), 29.r and 37.r (which may update $\mathit{Reset}[p.\mathit{node}].\mathit{rnd}$), 33.$r$, 35.$r$, and 36.$r$ (which may update $\mathsf{Dist}[p.y.\mathit{rnd}]$), and 34.$r$ (which may change the value of *Check*), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements 1.p, 3.p, 16.p, and 28.p. By (I60), statements 29.r, 33.r, 34.r, and 37.r cannot be executed while the antecedent holds. Statements 36.r falsifies the antecedent by (I60).

Statement 35.r may update $\mathsf{Dist}[p.y.\mathit{rnd}]$ only if executed when $\mathsf{Dist}[p.y.\mathit{rnd}] = \perp$, in which case it establishes $\mathsf{Dist}[p.y.\mathit{rnd}] = 2N$, by (I60) and (I37). This implies that the consequent holds. □

invariant $p@\{20..23\} \ \wedge\ \neg(\exists q :: q@\{34..36\}) \ \Rightarrow$

$$\begin{aligned}&\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd} \ \vee \\ &\mathsf{Dist}[p.y.\mathit{rnd}] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 3\end{aligned} \tag{I8}$$

Proof: The antecedent may be established only by statements 19.p and 36.q, where q is any process. Statement 19.p may establish the antecedent only if executed when $X[p.\mathit{node}] = p$ holds, in which case $\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ holds, by (I5).

Statement 36.q may establish the antecedent only if executed when $q@\{36\} \wedge p@\{20..23\}$ holds. If $\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ holds before the execution of 36.q, then it holds afterward as well, and thus (I8) is not falsified. So, assume that $q@\{36\} \wedge p@\{20..23\} \ \wedge\ \mathit{Reset}[p.\mathit{node}].\mathit{rnd} \neq p.y.\mathit{rnd}$ holds before the execution of 36.q. In this case, by (I7), $\mathsf{Dist}[p.y.\mathit{rnd}] > 0$ holds. Therefore, the function *Dequeue* decrements $\mathsf{Dist}[p.y.\mathit{rnd}]$ by 1. Moreover, by (I7), $\mathsf{Dist}[p.y.\mathit{rnd}] > 2N - 2 \cdot \big((\mathit{Check} - p - 1)$ **mod**

$N\big) - 2$ holds before $36.q$ is executed, which implies that $\mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 3$ holds afterward.

The consequent may be falsified only by statements $1.p$ and $3.p$ (which may update $p.node$), $16.p$ and $28.p$ (which may change the value of $p.y$), $29.r$ and $37.r$ (which may update $\mathit{Reset}[p.node].rnd$), $33.r$, $35.r$, and $36.r$ (which may update $\mathsf{Dist}[p.y.rnd]$), and $34.r$ (which may change the value of Check), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements $1.p$, $3.p$, $16.p$, and $28.p$. Statement $33.r$ establishes $r@\{34\}$, and hence falsifies the antecedent. By (I23), statement $29.r$ does not change the value of $\mathit{Reset}[p.node].rnd$. Statements $34.r$, $35.r$, and $36.r$ cannot be executed while the antecedent holds.

Finally, statement $37.r$ may falsify the consequent only if executed when $r@\{37\} \wedge \mathit{Reset}[p.node].rnd = p.y.rnd \wedge p.node = r.n$ holds. By (I24), this implies that $\mathit{Reset}[r.n].rnd = r.y.rnd$ holds. Also, by (I42), $\mathsf{Dist}[r.y.rnd] = 2N - 1$ holds. Combining these assertions, we have $p.y.rnd = r.y.rnd$, and hence $\mathsf{Dist}[p.y.rnd] = 2N - 1$. This implies that the consequent of (I8) holds after $37.r$ is executed. □

invariant $p@\{20..23\} \ \wedge \ q@\{37, 38\} \ \Rightarrow \ p.y.rnd \neq q.nextrnd$ (I9)

Proof: The antecedent may be established only by statements $19.p$ and $36.q$. Statement $19.p$ may establish the antecedent only if executed when $p@\{19\} \wedge X[p.node] = p \wedge q@\{37, 38\}$ holds. By (I5), this implies that $p.y.rnd \neq q.nextrnd$ holds. Thus, the consequent of (I9) is true after the execution of $19.p$.

Statement $36.q$ may establish the antecedent of (I9) only if executed when the antecedent of (I7) holds. By (I60), the third disjunct of (I14) does not hold before the execution of $36.q$. Thus, if $\mathit{Reset}[p.node].rnd = p.y.rnd$ holds before $36.q$ is executed, then $\mathsf{Dist}[p.y.rnd] = \perp \vee \mathsf{Dist}[p.y.rnd] = 2N$ holds. On the other hand, if $\mathit{Reset}[p.node].rnd \neq p.y.rnd$ holds before $36.q$ is executed, then by (I7), $\mathsf{Dist}[p.y.rnd] > 0$ holds. In either case, $\mathit{Dequeue}$ must return a value different from $p.y.rnd$. Hence, the consequent of (I9) is established.

The consequent may be falsified only by statements $16.p$ and $28.p$ (which may change the value of $p.y$) and $36.q$ (which may update $q.nextrnd$). However, $p@\{20..23\}$ is false after the execution of statements $16.p$ and $28.p$, and statement $36.q$ preserves (I9) as shown above. □

invariant $p@\{23\} \ \wedge \ Reset[p.node].rnd = p.y.rnd \ \Rightarrow$
$$Reset[p.node].free = false \qquad \text{(I10)}$$

Proof: (I10) may be falsified only by statements 22.p (which may establish $p@\{23\}$), 16.p and 28.p (which may change the value of $p.y.rnd$), 1.p and 3.p (which may change the value of $p.node$), and 29.q and 37.q (which may update $Reset[p.node]$), where q is any arbitrary process. However, $p@\{23\}$ is false after the execution of statements 1.p, 3.p, 16.p, and 28.p. Statement 22.p establishes the antecedent only if executed when $Reset[p.node] \neq p.y \ \wedge \ Reset[p.node].rnd = p.y.rnd$ holds, which implies that $Reset[p.node].free \neq p.y.free$. By (I28), $p.y.free = true$. Thus, $Reset[p.node].free = false$ holds.

If $q = p$, then each of 29.q and 37.q establishes $p@\{30, 38\}$, which implies that the antecedent is false.

Consider statements 29.q and 37.q, where $q \neq p$. Statement 29.q trivially establishes or preserves the consequent. Statement 37.q could potentially falsify (I10) only if executed when $p@\{23\} \wedge q@\{37\} \wedge q.n = p.node$ holds. In this case, by (I9), $p.y.rnd \neq Reset[p.node].rnd$ holds after the execution of 37.q. Thus, statement 37.q cannot falsify (I10). □

invariant $Y[i].free = true \ \Rightarrow \mathsf{Dist}[Y[i].rnd] = \bot \ \wedge$
$$(\forall p :: p@\{37, 38\} \ \Rightarrow \ Y[i].rnd \neq p.nextrnd) \qquad \text{(I11)}$$

Proof: The antecedent may be established only by statement 38.q, where q is any arbitrary process. However, by (I60) and (I40), if 38.q is executed when $q.n = i$ holds, then it establishes $\mathsf{Dist}[Y[i].rnd] = \bot \ \wedge \ \neg(\exists p :: p@\{37, 38\})$.

The consequent may be falsified only by statements 17.q, 26.q, and 38.q (which may update $Y[i].rnd$), 35.q (which may falsify $\mathsf{Dist}[Y[i].rnd] = \bot$), and 36.$q$ (which may update $q.nextrnd$, and may also establish $q@\{37, 38\}$), where q is any arbitrary process. Statements 17.q and 26.q falsify the antecedent. Statement 38.q preserves (I11) as shown above.

Statement 35.q may falsify $\mathsf{Dist}[Y[i].rnd] = \bot$ only if executed when $q@\{35\} \ \wedge \ Y[i].free = true \ \wedge \ q.y.rnd = Y[i].rnd$ holds. In this case, by (I26) and (I24), $Y[i].rnd = Reset[i].rnd$ and $Reset[q.n].rnd = q.y.rnd$ are both true as well. It follows that $Reset[q.n].rnd = Reset[i].rnd$ is also true, and hence $q.n = i$ holds, by (I15). By (I25), this in turn implies that $Y[i].free = false$ holds, a contradiction. It follows that statement 35.q cannot falsify the consequent while the antecedent holds.

If $Y[i].\mathit{free} = \mathit{false}$ holds before statement $36.q$ is executed, then it holds afterward, and hence (I11) is not falsified. If $Y[i] = \mathit{true}$ holds before $36.q$ is executed, then $\mathsf{Dist}[Y[i].\mathit{rnd}] = \bot$ holds as well, since (I11) is presumed to hold before the execution of $36.q$. Thus, $Y[i].\mathit{rnd}$ is not in the *Free* queue. This implies that a value different from $Y[i].\mathit{rnd}$ is dequeued, *i.e.*, $Y[i].\mathit{rnd} \neq q.\mathit{nextrnd}$ holds after the execution of $36.q$. □

invariant $p@\{37\} \;\Rightarrow\; (\forall i :: \mathit{Reset}[i].\mathit{rnd} \neq p.\mathit{nextrnd})$ (I12)

Proof: The antecedent may be established only by statement $36.p$, which may establish $\mathit{Reset}[i].\mathit{rnd} = p.\mathit{nextrnd}$ only if executed when $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 0$, *i.e.*, when $\mathit{Reset}[i].\mathit{rnd}$ is at the head of the *Free* queue. However, this is precluded by (I14).

The consequent may be falsified only by statement $36.p$ (which may update $p.\mathit{nextrnd}$), and statements $29.q$ and $37.q$ (which may update $\mathit{Reset}[i].\mathit{rnd}$), where q is any arbitrary process. However, statement $36.p$ preserves (I12) as shown above. By (I23), statement $29.q$ does not change the value of $\mathit{Reset}[i].\mathit{rnd}$. By (I60), the antecedent is false after the execution of statement $37.q$. □

invariant $p@\{38, 39\} \;\Rightarrow\; (\forall i :: \mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd})$ (I13)

Proof: The antecedent may be established only by statement $37.p$. Before its execution, $\mathit{Reset}[i].\mathit{rnd} \neq p.\mathit{nextrnd}$ holds, by (I12), and $\mathit{Reset}[p.n].\mathit{rnd} = p.y.\mathit{rnd}$ holds, by (I24). We consider two cases. First, suppose that $i = p.n$ holds before $37.p$ is executed. In this case, $p.y.\mathit{rnd} \neq p.\mathit{nextrnd}$ is true before the execution of $37.p$, and hence $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ is true after.

Second, suppose that $i \neq p.n$ holds before the execution of $37.p$. In this case, by (I15), $\mathit{Reset}[i].\mathit{rnd} \neq \mathit{Reset}[p.n].\mathit{rnd}$ holds as well. Because $\mathit{Reset}[p.n].\mathit{rnd} = p.y.\mathit{rnd}$ is true before $37.p$ is executed, we have $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ as well. Thus, $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ holds after the execution of $37.p$.

The consequent may be falsified only by statements $16.p$ and $28.p$ (which may update $p.y.\mathit{rnd}$), and $29.q$ and $37.q$ (which may update $\mathit{Reset}[i].\mathit{rnd}$), where q is any arbitrary process. However, the antecedent is false after the execution of $16.p$ and $28.p$, and by (I60), $29.q$ and $37.q$ are not enabled while the antecedent holds. □

invariant $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot \;\vee$
$(\exists p :: p@\{36\} \;\wedge\; p.n = i \;\wedge\; \mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 2N) \;\vee$
$(\exists p :: p@\{37\} \;\wedge\; p.n = i \;\wedge\; \mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 2N - 1)$ (I14)

Proof: The only statements that may falsify (I14) are $29.q$ (which may update $Reset[i].rnd$), $37.q$ (which may falsify $(\exists p :: p@\{36, 37\} \wedge p.n = i)$ and may update $Reset[i].rnd$), and $33.q$, $35.q$, and $36.q$ (which may update $\mathsf{Dist}[Reset[i].rnd]$), where q is any arbitrary process. By (I23), statement $29.q$ does not change the value of $Reset[i].rnd$.

Statement $37.q$ may falsify (I14) only if executed when $q.n = i$ holds, in which case it establishes $\mathsf{Dist}[Reset[i].rnd] = \bot$, by (I40).

If $33.q$ is enabled, then by (I60), $\neg(\exists p :: p@\{36, 37\})$ holds. Since (I14) is presumed to hold before the execution of $33.q$, this implies that $\mathsf{Dist}[Reset[i].rnd] = \bot$ holds both before and after $33.q$ is executed.

Statement $35.q$ may falsify $\mathsf{Dist}[Reset[i].rnd] = \bot$ only if executed when $q@\{35\} \wedge q.y.rnd = Reset[i].rnd$ holds. In this case, by (I24), $Reset[q.n].rnd = q.y.rnd$ holds as well. This implies that $Reset[q.n].rnd = Reset[i].rnd$ is true, and hence $q.n = i$ holds, by (I15). Because the *Enqueue* procedure puts $q.y.rnd$ at the tail of the *Free* queue, by (I37) and (I60), $35.q$ establishes $\mathsf{Dist}[q.y.rnd] = 2N$. Therefore, if statement $35.q$ falsifies $\mathsf{Dist}[Reset[i].rnd] = \bot$, then it establishes the second disjunct of (I14).

Statement $36.q$ may falsify (I14) only if executed when $q@\{36\} \wedge q.n = i \wedge \mathsf{Dist}[Reset[i].rnd] = 2N$ holds. In this case, the *Dequeue* decrements $\mathsf{Dist}[Reset[i].rnd]$ by one, establishing the third disjunct of (I14). □

invariant $i \neq h \Rightarrow Reset[i].rnd \neq Reset[h].rnd$ (I15)

Proof: The only statements that may falsify (I15) are $29.p$ and $37.p$, where p is any arbitrary process. By (I23), statement $29.p$ does not change the value of $Reset[i].rnd$, and hence cannot falsify (I15). Statement $37.p$ may falsify (I15) only if $p.n = h \wedge Reset[i].rnd = p.nextrnd$ holds prior to its execution. However, this is precluded by (I12). □

invariant $p@\{20..23\} \wedge q@\{33\} \wedge (Check = p) \Rightarrow$
$$Reset[p.node].rnd = p.y.rnd \vee q.usedrnd = p.y.rnd \qquad \text{(I16)}$$

Proof: The antecedent may be established only by statements $19.p$, $32.q$, and $34.r$, where r is any arbitrary process. However, by (I60), statement $34.r$ cannot be executed while $q@\{33\}$ holds.

Statement 19.p may establish $p@\{20\}$ only if executed when $X[p.node] = p$ holds. By (I5), this implies that $Reset[p.node].rnd = p.y.rnd$ holds both before and after 19.p is executed.

Statement 32.q may establish the antecedent only if executed when $p@\{20..23\} \wedge q@\{32\} \wedge Check = p$ holds. By (I30) and (I39), this implies that both $Inuse[p] = p.y.rnd$ and $q.ptr = p$ are also true. Thus, statement 32.q establishes $q.usedrnd = p.y.rnd$, and hence does not falsify (I16).

The consequent may be falsified only by statements 1.p, 3.p, 16.p, and 28.p (which may update either $p.node$ or $p.y.rnd$), 32.q (which may change the value of $q.usedrnd$), and 29.r and 37.r (which may update $Reset[p.node].rnd$), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements 1.p, 3.p, 16.p, and 28.p. Statement 32.q preserves (I16), as shown above. By (I60), Statements 29.r and 37.r cannot be executed while $q@\{33\}$ holds. □

invariant $A(p,i) \wedge A(q,i) \wedge p \neq q \Rightarrow$

$$\neg\big[p@\{17..20\} \wedge \big(q@\{3..8, 17..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i)\big)\big] \qquad \text{(I17)}$$

Proof: By Lemma A.1, the only statement that can establish $A(p,i)$ is 16.p. The only statements that may falsify the consequent while the antecedent holds are 16.p (which may establish $p@\{17..20\}$), and 16.q and 23.q (which may establish $q@\{3..8, 17..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i)$). Note that if statements 16.q and 23.q establish $q@\{3\}$, then they also establish $q.dir \neq stop$, which implies that $A(q,i)$ is false. Therefore, (I17) could potentially be falsified only if either 16.p or 16.q is executed, establishing $p@\{17\}$ or $q@\{17\}$, respectively. Without loss of generality, it suffices to consider only statement 16.p.

If $A(q,i) \wedge q@\{17..19\}$ holds before 16.p is executed, where $q \neq p$, then by the definition of $A(q,i)$, $X[i] = q$ holds as well. This implies that $X[i] \neq p$ holds both before and after 16.p is executed. Hence, $A(p,i)$ is false after the execution of 16.p.

Next, suppose that $A(q,i) \wedge \big(q@\{3..8, 20..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i)\big)$ holds before 16.p is executed, where $q \neq p$. In this case, by (I1), $Y[i].free = false$. This implies that 16.p does not establish $p@\{17\}$. □

invariant $A(p,i) \wedge A(q,i) \wedge p \neq q \Rightarrow$

$$\neg(p@\{3..14, 21, 22, 24..41\} \wedge q@\{3..14, 21, 22, 24..41\}) \qquad \text{(I18)}$$

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is 16.p. However, if 16.p establishes $A(p, i)$, it also establishes $p@\{17\}$, and hence it cannot falsify (I18). A similar argument applies to 16.q.

By symmetry, when considering statements that might falsify the consequent, it suffices to consider only 16.p, 19.p, 20.p, and 23.p (which may establish $p@\{3..14, 21, 22, 24..41\}$). Statements 16.$p$, 19.$p$, and 23.$p$ can establish $p@\{3..14, 21, 22, 24..41\}$ only by establishing $p@\{3\}$, in which case they also establish $p.dir \neq stop$. This implies that $A(p, i)$ is false. Thus, these statements cannot falsify (I18). Similar reasoning applies if statement 20.p establishes $p@\{3\}$.

Statement 20.p could also establish $p@\{3..14, 21, 22, 24..41\}$ by establishing $p@\{21\}$. In this case, we have $Acquired[i] = false$ before its execution. If $A(p, i)$ is false before 20.p is executed, then by Lemma A.1, it is also false afterward, and hence (I18) is not falsified. So, suppose that $A(p, i)$ is true before 20.p is executed. By (I17), this implies that $A(q, i) \wedge q@\{3..8, 21, 22, 24\}$ is false. Thus, 20.p could potentially falsify (I18) only if executed when $A(q, i) \wedge q@\{9..14, 25..41\}$ holds. However, in this case, by (I4), we have $Acquired[i] = true$. Thus, statement 20.p cannot falsify (I18). □

invariant $A(p, i) \wedge q.n = i \Rightarrow \neg(p@\{17..19\} \wedge q@\{28..38\})$ (I19)

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is 16.p, which may do so only if $Y[i].free = true$. By (I25), this implies that $q@\{28..38\} \wedge q.n = i$ is false. Therefore, statement 16.p cannot falsify (I19). Any statement that updates $q.n$ also establishes $q@\{9, 15\}$, and hence cannot falsify (I19).

The only other statement that may falsify (I19) is 27.q (which establishes $q@\{28..38\}$), which may do so only if executed when $q.n = i$ holds. In this case, it falsifies $X[i] = p$, which implies that $A(p, i) \wedge p@\{17..19\}$ is false as well. □

invariant $A(p, i) \wedge q.n = i \Rightarrow \neg(p@\{20, 21\} \wedge q@\{30..38\})$ (I20)

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is 16.p, which establishes $p@\{17, 3\}$. Hence, it cannot falsify (I20). Any statement that updates $q.n$ also establishes $q@\{9, 15\}$, and hence cannot falsify (I20).

The only other statements that may falsify (I20) are 19.p (which may establish $p@\{20, 21\}$) and 29.q (which establishes $q@\{30..38\}$). Statement 19.p may falsify (I20) only by establishing $p@\{20\}$, which it does only if $X[p.node] = p$. By (I5), this implies that $p@\{19\} \wedge X[p.node] = p \wedge Reset[p.node] = p.y$ holds before its execution. Thus, $19.p$ may potentially falsify (I20) only if executed when $A(p, p.node)$ holds. If $i \neq p.node$, then $A(p, i)$ is clearly false both before and after the execution of $19.p$. If $i = p.node$, then $A(p, i)$ holds before $19.p$ is executed, which implies that $q@\{30..38\} \wedge q.n = i$ is false, by (I19). Thus, statement $19.p$ cannot falsify (I20).

Statement $29.q$ may falsify (I20) only if executed when $q.n = i \wedge p@\{20, 21\}$ holds. By (I28), this implies that $p.y.free = true$ holds. Because statement $29.q$ establishes $Reset[i].free = false$, this implies that $p@\{20, 21\} \wedge Reset[i] \neq p.y$ is established, *i.e.*, $A(p, i)$ is false after its execution. □

invariant $A(p, i) \wedge q.n = i \Rightarrow \neg(p@\{3..10, 22, 24\} \wedge q@\{31..38\})$ (I21)

Proof: By Lemma A.1, the only statement that can establish $A(p, i)$ is $16.p$. However, if $16.p$ establishes $A(p, i)$, then it also establishes $p@\{17\}$, and hence cannot falsify (I21). Any statement that updates $q.n$ also establishes $q@\{9, 15\}$, and hence cannot falsify (I21).

The only other statements that may falsify (I21) are $16.p$, $19.p$, $20.p$, $21.p$, and $23.p$ (which may establish $p@\{3..10, 22, 24\}$), and $30.q$ (which may establish $q@\{31..38\}$).

Statements $16.p$, $19.p$, $20.p$, and $23.p$ can establish $p@\{3..10, 22, 24\}$ only by establishing $p@\{3\}$, in which case they also establish $p.dir \neq stop$. This implies that $A(p, i)$ is false. Statement $21.p$ can neither establish nor falsify $A(p, i)$. Thus, it may falsify (I21) only if executed when $A(p, i) \wedge q.n = i \wedge q@\{31..38\}$ holds, but this is precluded by (I20).

Statement $30.q$ may falsify (I21) only if executed when

$$q@\{30\} \wedge A(p, i) \wedge p@\{3..10, 22, 24\} \wedge q.n = i \wedge \\ (q.dir = stop \vee Round[q.y.rnd] = false)$$

holds. By (I60), $p@\{8..10\}$ and $q@\{30\}$ cannot hold simultaneously. So, assume that the following assertion holds prior to the execution of $30.q$.

$$q@\{30\} \wedge A(p, i) \wedge p@\{3..7, 22, 24\} \wedge q.n = i \wedge \\ (q.dir = stop \vee Round[q.y.rnd] = false) \qquad \text{(A.2)}$$

If $q.dir = stop$ holds, then by (I33), $q.n = q.node$. Because $q.n = i$, this implies that $A(q, i)$ holds. By (A.2), this implies that $A(p, i) \wedge A(q, i) \wedge p@\{3..7, 22, 24\} \wedge q@\{30\}$ holds. However, this is precluded by (I18).

The only other possibility is that

$$q@\{30\} \ \wedge \ A(p,i) \ \wedge \ p@\{3..7, 22, 24\} \ \wedge \ q.n = i \ \wedge \ \mathit{Round}[q.y.rnd] = \mathit{false}$$

holds before $30.q$ is executed. In this case, $\mathit{Reset}[q.n].rnd = q.y.rnd$ holds, by (I24), and $\mathit{Reset}[i].rnd = p.y.rnd$ also holds, by the definition of $A(p,i)$ and (I3). Thus, $q.y.rnd = p.y.rnd$. In addition, by (I2), $\mathit{Round}[p.y.rnd] = \mathit{true}$. Thus, $\mathit{Round}[q.y.rnd] = \mathit{true}$, which is a contradiction. □

invariant $A(p,i) \ \wedge \ \big(p@\{3..8, 20..22, 24\} \ \vee \ (p@\{9, 25..38\} \ \wedge \ p.n = i)\big) \ \wedge$
$q@\{20..23\} \ \wedge \ p \neq q \ \Rightarrow$
$p.y.rnd \neq q.y.rnd$ (I22)

Proof: The only statements that may falsify the consequent are $16.p$ and $28.p$ (which may change the value of $p.y.rnd$) and $16.q$ and $28.q$ (which may change the value of $q.y.rnd$). However, the antecedent is false after the execution of $16.q$ and $28.q$. Also, $16.p$ either establishes $p@\{17\}$ or $p@\{3\} \wedge p.dir = \mathit{right}$. The latter implies that $A(p,i)$ is false. Thus, statement $16.p$ cannot falsify (I22).

Statement $28.p$ may falsify (I22) only if executed when $A(p,i) \ \wedge \ p.n = i$ holds. In this case, $\mathit{Reset}[i].rnd = p.y.rnd$ holds, by (I3). It follows that statement $28.p$ does not change the value of $p.y.rnd$ if executed when the antecedent holds.

The antecedent may be established only by statements $16.p$ (which, by Lemma A.1, may establish $A(p,i)$ and may also establish $p@\{3..8, 20..22, 24\}$), $19.p$ and $23.p$ (which may establish $p@\{3..8, 20..22, 24\}$), and $19.q$ (which may establish $q@\{20..23\}$). However, as shown above, statement $16.p$ cannot falsify (I22).

Statement $19.q$ can establish $q@\{20\}$ only if $q@\{19\} \ \wedge \ X[q.node] = q$ holds. If $i = q.node$, then by (I5), $A(q,i)$ holds as well. However, by (I17) (with p and q exchanged), this implies that the antecedent of (I22) is false. Thus, $19.q$ cannot falsify (I22) in this case.

On the other hand, if $q@\{19\} \ \wedge \ X[q.node] = q \ \wedge \ i \neq q.node$ holds, then we have $q.y = \mathit{Reset}[q.node]$, by (I5), and $\mathit{Reset}[q.node].rnd \neq \mathit{Reset}[i].rnd$, by (I15). If the antecedent of (I22) is true, then we have either $A(p,i) \ \wedge \ p@\{3..8, 20..22, 24\} \ \vee$ $(p@\{9, 25..37\} \ \wedge \ p.n = i)$ or $p@\{38\}$. In the former case, by (I3) and the definition of $A(p,i)$, we have $p.y.rnd = \mathit{Reset}[i].rnd$. In the latter case, by (I13), we have $p.y.rnd \neq \mathit{Reset}[q.node].rnd$. Thus, in either case, $p.y.rnd \neq q.y.rnd$ holds. Hence, statement $19.q$ cannot falsify (I22).

Statements $19.p$ and $23.p$ are the remaining statements to consider. Statement $23.p$ establishes $p@\{3\} \ \wedge \ p.dir = \mathit{left}$, which implies that $A(p, i)$ is false. Statement $19.p$ establishes either $p@\{3\} \ \wedge \ p.dir = \mathit{left}$ or $p@\{20\}$. In the former case, $A(p, i)$ is false. In the latter case, note that statement $19.p$ may falsify (I22) only if executed when $q@\{20..23\}$ holds. In addition, by Lemma A.1, $19.p$ can establish $A(p, i) \ \wedge \ p@\{20\}$ only if executed when $A(p, i) \ \wedge \ p@\{19\} \ \wedge \ X[i] = p$ is true. By (I28) and the definition of $A(p, i)$, we therefore have the following prior to the execution of $19.p$.

$$q@\{20..23\} \ \wedge \ A(p, i) \ \wedge \ p@\{19\} \ \wedge \ X[i] = p \ \wedge \ p.y = \mathit{Reset}[i] \ \wedge \ \mathit{Reset}[i].\mathit{free} = \mathit{true} \tag{A.3}$$

By applying (I6), (I7), and (I8) to $q@\{20..23\}$, we also have either $q.y.\mathit{rnd} = \mathit{Reset}[q.\mathit{node}].\mathit{rnd}$ or $\mathsf{Dist}[q.y.\mathit{rnd}] \neq \bot$. This gives us two cases to analyze.

- $q.y.\mathit{rnd} = \mathit{Reset}[q.\mathit{node}].\mathit{rnd}$. Note that statement $19.p$ can falsify (I22) only if executed when the following holds.

  $$p.y.\mathit{rnd} = q.y.\mathit{rnd} \tag{A.4}$$

  By (A.3), (A.4), and our assumption that $q.y.\mathit{rnd} = \mathit{Reset}[q.\mathit{node}].\mathit{rnd}$ holds, we have $\mathit{Reset}[i].\mathit{rnd} = \mathit{Reset}[q.\mathit{node}].\mathit{rnd}$. By (I15), this implies that $i = q.\mathit{node}$.

  If $q@\{20..22\}$ holds before the execution of $19.p$, then by (A.3) and (I17), $A(q, i)$ is false. By the definition of $A(q, i)$ this implies that $\mathit{Reset}[i] \neq q.y$. By (A.3), this implies that $p.y \neq q.y$. In addition, by (I28), we have $q.y.\mathit{free} = \mathit{true}$. By (A.3), this implies that $p.y.\mathit{rnd} \neq q.y.\mathit{rnd}$, which contradicts (A.4).

  On the other hand, if $q@\{23\}$ holds before the execution of $19.p$, then we have $\mathit{Reset}[q.\mathit{node}].\mathit{free} = \mathit{false}$, by (I10). Because $i = q.\mathit{node}$, this contradicts (A.3).

- $\mathsf{Dist}[q.y.\mathit{rnd}] \neq \bot$. By Lemma A.1, statement $19.p$ may establish the antecedent only if $A(p, i)$ holds before its execution. By (I19), this implies that $\neg(\exists r :: r@\{36, 37\} \ \wedge \ r.n = i)$ holds. Hence, by (I14), $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot$ holds as well. By (A.3), we therefore have $\mathsf{Dist}[p.y.\mathit{rnd}] = \bot$. Because $\mathsf{Dist}[p.y.\mathit{rnd}] = \bot$ and $\mathsf{Dist}[q.y.\mathit{rnd}] \neq \bot$ both hold, we have $p.y.\mathit{rnd} \neq q.y.\mathit{rnd}$. It follows that statement $19.p$ cannot falsify (I22). $\square$

invariant $p@\{29\} \ \Rightarrow \ \mathit{Reset}[p.n] = p.y$ (I23)

invariant $p@\{30..37\} \ \Rightarrow \ \mathit{Reset}[p.n] = (\mathit{false}, p.y.\mathit{rnd})$ (I24)

invariant $p@\{27..38\} \;\Rightarrow\; Y[p.n] = (\mathit{false}, 0)$ (I25)

invariant $Y[i].\mathit{free} = \mathit{true} \;\Rightarrow\; Y[i] = \mathit{Reset}[i]$ (I26)

invariant $\mathit{Reset}[i].\mathit{rnd} \neq 0$ (I27)

invariant $p@\{17..23\} \;\Rightarrow\; p.y.\mathit{free} = \mathit{true}$ (I28)

invariant $p@\{17..23, 29..40\} \;\Rightarrow\; p.y.\mathit{rnd} \neq 0$ (I29)

invariant $p@\{19..23\} \;\Rightarrow\; \mathit{Inuse}[p] = p.y.\mathit{rnd}$ (I30)

Proof: By (I60), all writes to *Reset* (statements 29 and 37) and to Y, except for statement 17 (statements 26 and 38), and all operations involving the *Free* queue (statements 33, 35, and 36) occur within mutually exclusive regions of code. Given this and the initial condition $(\forall i, p :: Y[i] = (\mathit{true}, i) \;\wedge\; \mathit{Reset}[i] = (\mathit{true}, i)) \;\wedge\; \mathit{Free} = (T+1) \rightarrow \cdots \rightarrow S$, each of these invariants easily follows. Note that statement 17 establishes $Y[i] = (\mathit{false}, 0)$, and hence cannot falsify either (I25) or (I26). Note also that (I25) implies that statements 29 and 37 cannot falsify (I26). □

invariant $p@\{2, 3, 15..24\} \;\Rightarrow\; (0 \leq p.\mathit{level} \leq L) \;\wedge\; (1 \leq p.\mathit{node} \leq T)$ (I31)

invariant $p@\{9, 25..40\} \;\Rightarrow$

$$1 \leq p.n \leq T \;\wedge\; p.n = p.\mathit{path}[p.j].\mathit{node} \;\wedge$$
$$p.\mathit{dir} = p.\mathit{path}[p.j].\mathit{dir} \;\wedge\; p.j = \mathsf{lev}(p.n) \qquad \text{(I32)}$$

invariant $p@\{9, 25..40\} \;\wedge\; p.\mathit{dir} = \mathit{stop} \;\Rightarrow\; p.n = p.\mathit{node}$ (I33)

invariant $p@\{9, 25..40\} \;\wedge\; p.n = p.\mathit{node} \;\Rightarrow\; p.\mathit{dir} = \mathit{stop}$ (I34)

invariant $p@\{2..41\} \;\Rightarrow\; (0 \leq p.\mathit{level} \leq L+1) \;\wedge\; (\mathsf{lev}(p.\mathit{node}) = p.\mathit{level})$ (I35)

invariant $p@\{39, 40\} \;\Rightarrow\; p.n = p.\mathit{node}$ (I36)

Proof: These invariants easily follow from the program text and structure of the renaming tree. □

invariant $\neg(\exists p :: p@\{36\}) \;\Rightarrow\; \|\mathit{Free}\| = 2N$ (I37)

invariant $(\exists p :: p@\{36\}) \;\Rightarrow\; \|\mathit{Free}\| = 2N+1$ (I38)

invariant $p@\{32..34\} \;\Rightarrow\; p.\mathit{ptr} = \mathit{Check}$ (I39)

invariant $p@\{37, 38\} \;\Rightarrow\; \mathsf{Dist}[p.\mathit{nextrnd}] = \perp$ (I40)

invariant $p@\{36\} \;\Rightarrow\; \mathsf{Dist}[p.y.\mathit{rnd}] = 2N$ (I41)

invariant $p@\{37\} \;\Rightarrow\; \mathsf{Dist}[p.y.\mathit{rnd}] = 2N-1$ (I42)

Proof: Note that every statement that accesses the *Free* queue (statements 33, 35, and 36) executes within a mutually exclusive region of code. From this and the initial condition, $\|Free\| = 2N$, these invariants easily follow. Note also that by (I24), if $p@\{35\}$ holds, then $p.y.rnd = Reset[p.n].rnd$ holds. By (I14) and (I60), this in turn implies that $\mathsf{Dist}[p.y.rnd] = \bot$ holds. Hence, statement $35.p$ does not enqueue a duplicate entry onto the *Free* queue. $\Box$

invariant (Exclusion) $\big|\{p :: p@\{8..10, 13\}\}\big| \le 1$ (I60)

Proof: From the specification of the `ENTRY` and `EXIT` routines, (I60) could be falsified only if two processes p and q stop at the same splitter in the renaming tree, *i.e.*, we have $p \neq q \ \wedge\ p@\{4, 5, 8..11\} \ \wedge\ q@\{4, 5, 8..11\} \ \wedge\ p.node = q.node$. However, this is precluded by (I18). $\Box$

A.3 Proof of Adaptivity under the RMR Measure

The remaining invariants are needed to establish adaptivity (I61). These invariants formalize the following rather intuitive reasoning: for a process p to reach splitter i in the renaming tree by moving right (left) from i's parent, some other process must have either stopped or moved left (right) at i's parent. From this, it follows that the depth to which p descends in the renaming tree is proportional to the point contention that p experiences.

invariant $(1 \le i \le S) \ \wedge\ Round[i] = true \ \Rightarrow$

$$\begin{aligned}&\big(\exists p :: p.y.rnd = i \ \wedge \\ &\quad \big(p@\{3..5, 8, 22..24\} \ \vee\ (p@\{9, 25..39\} \ \wedge\ p.n = p.node)\big) \ \wedge \\ &\quad \big(p@\{3..5, 8\} \ \Rightarrow\ (p.dir = stop \ \wedge\ 1 \le p.node \le T)\big)\big)\end{aligned} \qquad \text{(I43)}$$

Proof: The antecedent may be established only by statement $21.p$, which also establishes the consequent.

Suppose that

$$\begin{aligned}&p.y.rnd = i \ \wedge\ \big(p@\{3..5, 8, 22..24\} \ \vee\ (p@\{9, 25..39\} \ \wedge\ p.n = p.node)\big) \ \wedge \\ &\big(p@\{3..5, 8\} \ \Rightarrow\ (p.dir = stop \ \wedge\ 1 \le p.node \le T)\big)\end{aligned} \qquad \text{(A.5)}$$

holds. We consider each condition separately. The only statements that may falsify $p.y.rnd = i$ are $16.p$ and $28.p$. Statement $16.p$ cannot be executed while (A.5) holds. If statement $28.p$ is executed while (A.5) holds, then by (I3), it does not change the value of $p.y.rnd$.

The only statements that may falsify $p@\{3..5, 8, 22..24\}$ are $3.p$ and $8.p$. If statement $3.p$ is executed while (A.5) holds, then by (I35), we also have $p.level \leq L$, and hence statement $3.p$ establishes $p@\{4\}$. Similarly, if statement $8.p$ is executed while (A.5) holds, then it establishes $p@\{9\} \wedge p.n = p.node$. Thus, these statements cannot falsify (A.5).

The only statements that may falsify $p@\{9, 25..39\} \wedge p.n = p.node$ are $25.p$, $30.p$, $38.p$, and $39.p$. By (I34), statements $25.p$, $30.p$, and $38.p$ establish $p@\{26, 31, 39\}$. If $p.y.rnd = i$, then statement $39.p$ falsify the antecedent of (I43).

The only statements that may falsify $p@\{3..5, 8\} \Rightarrow (p.dir = stop \wedge 1 \leq p.node \leq T)$ are $1.p$ and $3.p$ (which may update $p.node$), $7.p$ (which establishes $p@\{3..5, 8\}$), and $16.p$, $19.p$, $20.p$, $23.p$, and $24.p$ (which may establish $p@\{3..5, 8\}$ and also update $p.dir$). Statements $1.p$, $7.p$, $16.p$, $19.p$, and $20.p$ cannot be executed while (A.5) holds. If statement $3.p$ is executed while $p.dir = stop$ holds, then it does not update $p.node$. If $p.y.rnd = i$, then statement $23.p$ falsifies the antecedent of (I43). By (I31), statement $24.p$ establishes $p.dir = stop \wedge 1 \leq p.node \leq T$, and hence preserves (A.5). □

invariant $p@\{4..14, 25..41\} \wedge (p.node \leq T) \Rightarrow p.path[p.level] = (p.node, stop)$ (I44)

Proof: The antecedent of (I44) can only be established by statement $3.p$. By (I31), it does so only if $p.level = L \vee p.dir = stop$ holds prior to its execution. If $p.dir = stop$ holds, then $3.p$ establishes the consequent of (I44). If $p.level = L \wedge p.dir \neq stop$ holds before $3.p$ is executed, then by (I35), $p.node > T$ holds afterward. No statement can falsify the consequent of (I44) while the antecedent holds. □

invariant $p@\{2..41\} \wedge (2i \xrightarrow{*} p.node) \Rightarrow p.path[\mathsf{lev}(i)] = (i, left)$ (I45)

Proof: The only statements that may falsify (I45) are $1.p$ and $3.p$. Statement $1.p$ establishes $p.node = 1$, which implies that $2i \xrightarrow{*} p.node$ is false. Statement $3.p$ may establish $2i \xrightarrow{*} p.node$ only if it also establishes $2i = p.node$, which can happen only if it is executed when $p.node = i \wedge p.dir = left$ holds. In this case, by (I35), statement $3.p$ establishes the consequent. □

invariant $p@\{3\} \ \wedge\ (p.level > 0) \ \Rightarrow$
$\quad p.path[0].node = 1 \ \wedge$
$\quad (\forall l : 0 \leq l < p.level - 1 :: p.path[l].node \xrightarrow{*} p.path[l+1].node) \ \wedge$
$\quad p.path[p.level - 1].node \xrightarrow{*} p.node$ (I46)

invariant $p@\{26..38\} \ \Rightarrow\ (p.n = p.node) \ \vee\ (2 \cdot p.n \xrightarrow{*} p.node)$ (I47)

Proof: Each iteration of the **repeat** loop of statements 2–3 descends one level in the renaming tree, starting with splitter 1 (the root). When descending from level l, $p.path[l]$ is updated (statement 3) to indicate the splitter visited at level l. From this, invariant (I46) easily follows.

The **repeat** loop of statements 2–3 terminates only if $p.level \geq L \ \vee\ p.dir = stop$ holds prior to the execution of statement $3.p$. If $p.dir = stop$ holds when $3.p$ is executed, then when p executes within statements 4–14 and 25–41, $p.node$ equals the splitter at which it stopped. If $p.level \geq L \ \wedge\ p.dir \neq stop$ holds when $3.p$ is executed, then when p executes within statements 4–14 and 25–41, $p.node$ equals a splitter at level $L + 1$ (in which case there is no actual splitter corresponding to $p.node$). In either case, the **for** loop at statement 9 will ascend the renaming tree, visiting only $p.node$ and its ancestors. The corresponding splitter is indicated by the variable $p.n$. Moreover, if p moved right while descending the renaming tree, then p returns from *ReleaseNode* at statement 25. If p stopped at that splitter, then $p.n = p.node$ holds. If p moved left from that splitter, then $2 \cdot p.n \xrightarrow{*} p.node$ holds. From these observations, it should be clear that (I47) is an invariant. $\square$

invariant $p@\{0, 1\} \ \Rightarrow\ \mathsf{Loc}[p] = 0 \ \wedge\ \mathsf{Loc}[p + N] = 0$ (I48)

invariant $p@\{2, 15\} \ \Rightarrow\ \mathsf{Loc}[p] = p.node \ \wedge\ \mathsf{Loc}[p + N] = p.node$ (I49)

invariant $p@\{16\} \ \Rightarrow$
$\quad \big(X[p.node] = p \ \Rightarrow\ \mathsf{Loc}[p] = p.node \ \wedge\ \mathsf{Loc}[p + N] = p.node\big) \ \wedge$
$\quad \big(X[p.node] \neq p \ \Rightarrow\ \mathsf{Loc}[p] = 2 \cdot p.node \ \wedge\ \mathsf{Loc}[p + N] = 2 \cdot p.node\big)$ (I50)

invariant $p@\{17..24\} \ \Rightarrow$
$\quad \big(A(p, p.node) \ \Rightarrow\ \mathsf{Loc}[p] = p.node \ \wedge\ \mathsf{Loc}[p + N] = p.node\big) \ \wedge$
$\quad \big(\neg A(p, p.node) \ \Rightarrow\ \mathsf{Loc}[p] = 2 \cdot p.node \ \wedge\ \mathsf{Loc}[p + N] = 2 \cdot p.node\big)$ (I51)

invariant $p@\{3\} \ \Rightarrow$
$\quad \big(p.dir = stop \ \Rightarrow\ \mathsf{Loc}[p] = p.node \ \wedge\ \mathsf{Loc}[p + N] = p.node\big) \ \wedge$
$\quad \big(p.dir = left \ \Rightarrow\ \mathsf{Loc}[p] = 2 \cdot p.node \ \wedge\ \mathsf{Loc}[p + N] = 2 \cdot p.node\big) \ \wedge$
$\quad \big(p.dir = right \ \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node + 1 \ \wedge$

$$\mathsf{Loc}[p+N] = 2 \cdot p.node + 1\big) \tag{I52}$$

invariant $p@\{4..14, 25..41\} \;\Rightarrow\; \mathsf{Loc}[p+N] = p.node$ (I53)

invariant $p@\{4..8\} \;\Rightarrow\; \mathsf{Loc}[p] = p.node$ (I54)

invariant $p@\{9, 25\} \;\Rightarrow\; p.n \xrightarrow{*} \mathsf{Loc}[p]$ (I55)

invariant $p@\{26..40\} \;\Rightarrow\; \mathsf{Loc}[p] = p.n$ (I56)

Proof: These invariants easily follow from the program text and the structure of the renaming tree. Note that $p@\{16\}$ is established only if $X[p.node] = p$ holds. Also note that whenever $X[p.node] = p$ is falsified by either $15.q$ or $27.q$, where q is any arbitrary process, q also establishes $\mathsf{Loc}[p] = 2 \cdot p.node \;\wedge\; \mathsf{Loc}[p+N] = 2 \cdot p.node$.

Statement $16.p$ may establish $p@\{17\}$ only if executed when $Y[p.node].free = true$ holds. By (I26), this implies that $Y[p.node] = Reset[p.node]$ holds. From this condition, the consequent of (I50), and the definition of $A(p, i)$, it follows that if statement $16.p$ establishes $p@\{17\}$, then the consequent of (I51) is true.

$A(p, p.node)$ potentially could be falsified by some process $q \neq p$ only by executing one of the statements $15.q$, $27.q$, $29.q$, or $37.q$. However, by (I24), $q@\{37\}$ implies that $Reset[q.n].free = false$ holds. Moreover, by (I28), $p@\{17..22\}$ implies $p.y.free = true$. It follows that statement $37.q$ cannot change the value of $A(p, p.node)$ from true to false. If one of $15.q$, $27.q$, and $29.q$ falsifies $A(p, p.node)$, then it also assigns the value $2 \cdot p.node$ to each of $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$. □

invariant $Y[i].free = false \;\Rightarrow$

$$\begin{aligned}
\mathcal{A}\colon\ & \big(\exists p :: \big(p@\{3..8, 18..24\} \;\vee\; (p@\{9, 25..38\} \;\wedge\; p.n = i)\big) \;\wedge \\
& \quad (p.node = i) \;\wedge\; (p@\{3\} \;\Rightarrow\; p.dir \neq right)\big) \;\vee \\
\mathcal{B}\colon\ & \big(\exists p :: p@\{2..9, 15..40\} \;\wedge\; (2i \xrightarrow{*} p.node) \;\wedge \\
& \quad (p@\{9, 25..40\} \;\Rightarrow\; 2i \xrightarrow{*} p.n)\big) \;\vee \\
\mathcal{C}\colon\ & \big(\exists p :: p@\{9, 25..38\} \;\wedge\; p.n = i \;\wedge\; (p@\{9, 25\} \;\Rightarrow\; 2i \xrightarrow{*} \mathsf{Loc}[p])\big)
\end{aligned} \tag{I57}$$

Proof: Note that we may assume $1 \leq i \leq T$, because $Y[i]$ is defined only for values of i in this range.

The only statements that can establish the antecedent are $17.q$ and $26.q$, where q is any arbitrary process. Statement $17.q$ establishes disjunct $\mathcal{A}$. If statement $26.q$ establishes the antecedent, then disjunct $\mathcal{C}$ holds both before and after its execution. We now consider statements that may falsify each of the three disjuncts of the consequent.

Disjunct $\mathcal{A}$: Suppose that the following assertion holds.

$$\begin{aligned}&\big(p@\{3..8, 18..24\} \ \vee \ (p@\{9, 25..38\} \ \wedge \ p.n = i)\big) \ \wedge \\ &(p.node = i) \ \wedge \ (p@\{3\} \ \Rightarrow \ p.dir \neq right)\end{aligned} \tag{A.6}$$

We consider each condition separately. The only statements that may falsify $p@\{3..8, 18..24\}$ are $3.p$ and $8.p$. If $3.p$ is executed while $p.node = i \ \wedge \ p.dir = stop$ holds, then it establishes $p@\{4\} \ \wedge \ p.node = i$, and hence preserves (A.6). On the other hand, if $3.p$ is executed while $p.node = i \ \wedge \ p.dir = left$ holds, then it establishes $p@\{2, 6\} \ \wedge \ p.node = 2i$, and hence disjunct $\mathcal{B}$ is established. If statement $8.p$ is executed while (A.6) holds, then since $i \leq T$, by (I35), $p.level \leq L$ holds before and after its execution. Thus, in this case, $8.p$ establishes $p@\{9\} \ \wedge \ p.n = i$. It follows that these statements cannot falsify (A.6).

The only statements that may falsify $p@\{9, 25..38\} \ \wedge \ p.n = i$ are $25.p$, $30.p$, and $38.p$. By (I34), statements $25.p$ and $30.p$ establish $p@\{26, 31\}$. If $p.n = i$, then statement $38.p$ falsifies the antecedent of (I57).

The only statements that may update $p.node$ are $1.p$ and $3.p$. Statement $1.p$ cannot be executed while (A.6) holds. Statement $3.p$ preserves (I57) as shown above.

The only statement that may falsify $p@\{3\} \ \Rightarrow \ p.dir \neq right$ is $16.p$, which cannot be executed while (A.6) holds.

Disjunct $\mathcal{B}$: Suppose that the following assertion holds.

$$p@\{2..9, 15..40\} \ \wedge \ (2i \xrightarrow{*} p.node) \ \wedge \ (p@\{9, 25..40\} \ \Rightarrow \ 2i \xrightarrow{*} p.n) \tag{A.7}$$

We consider each condition separately. The only statements that may falsify $p@\{2..9, 15..40\}$ are $25.p$, $30.p$, $38.p$, and $40.p$ (which may return from *ReleaseNode* and terminate the **for** loop at statement 9). However, since $2i \xrightarrow{*} p.n$ implies $\mathsf{lev}(p.n) > 1$, the **for** loop must iterate again, establishing $p@\{9\}$. If these statements preserve $2i \xrightarrow{*} p.n$, then (A.7) is preserved. Otherwise, p ascends the renaming tree by a level, and $p@\{9\} \ \wedge \ p.n = i$ is established. Moreover, by (I55) and (I56), $2i \xrightarrow{*} \mathsf{Loc}[p]$ holds before and after the execution of each of these statements. This implies that disjunct $\mathcal{C}$ holds.

The only statements that may update $p.node$ are $1.p$ and $3.p$. Statement $1.p$ cannot be executed while (A.7) holds. Statement $3.p$ only updates $p.node$ as the renaming tree is descended. Thus, it cannot falsify $2i \xrightarrow{*} p.node$.

The only statements that may falsify $p@\{9, 25..40\} \Rightarrow 2i \xrightarrow{*} p.n$ are $8.p$ (which may establish $p@\{9, 25..40\}$ and also update $p.n$) and $25.p$, $30.p$, $38.p$, and $40.p$ (which may return from *ReleaseNode* and update $p.n$). If statement $8.p$ establishes $p@\{9\}$, then it also establishes $p.n = p.node$. Thus, it cannot falsify (A.7). Statements $25.p$, $30.p$, $38.p$, and $40.p$ preserve (I57) as shown above.

Disjunct $\mathcal{C}$: Suppose that the following assertion holds.

$$p@\{9, 25..38\} \wedge p.n = i \wedge (p@\{9, 25\} \Rightarrow 2i \xrightarrow{*} \mathsf{Loc}[p]) \tag{A.8}$$

This assertion implies that $p@\{9, 25..38\}$ holds. Thus, it may be falsified only by statement $25.p$ (which updates $\mathsf{Loc}[p]$ and may falsify $p@\{9, 25..38\}$ — note that no other process can modify $\mathsf{Loc}[p]$ while $p@\{9, 25..38\}$ holds), and statements $30.p$ and $38.p$ (which may falsify $p@\{9, 25..38\}$, establish $p@\{9, 25\}$, or modify $p.n$). Statement $25.p$ may falsify $p@\{9, 25..38\}$ only if executed when $p.dir = right$ holds. By (I45) and (I55), $p@\{25\} \wedge 2i \xrightarrow{*} \mathsf{Loc}[p]$ implies $p.path[\mathsf{lev}(i)] = (i, \mathit{left})$ — informally, p moved left from splitter i when descending the renaming tree. Thus, by (I32), $p.dir = \mathit{left}$ holds before the execution of $25.p$. It follows that statement $25.p$ establishes $p@\{26\}$, and hence it cannot falsify (A.8). Statement $38.p$ falsifies the antecedent of (I57) if executed when $p.n = i$ holds.

Statement $30.p$ may falsify (A.8) only if executed when $Round[p.y.rnd] = true$ holds. So, assume that $p@\{30\} \wedge Round[p.y.rnd] = true$ holds. Then, by (I43), there exists a process q such that

$$\begin{aligned}&\big(q@\{3..5, 8, 22..24\} \vee (q@\{9, 25..39\} \wedge q.n = q.node)\big) \wedge \\ &\big(q@\{3..5, 8\} \Rightarrow (q.dir = stop \wedge 1 \le q.node \le T)\big) \wedge \\ &q.y.rnd = p.y.rnd.\end{aligned} \tag{A.9}$$

If $q = p$, then by (A.9) and our assumption that $p@\{30\}$ holds, we have $p.n = p.node$, which implies, by (I34), that statement $30.p$ establishes $p@\{31\}$, preserving (A.8).

So, suppose that $q \ne p$. By (I24), we have $Reset[i].rnd = p.y.rnd$, which by (A.9) implies that

$$Reset[i].rnd = q.y.rnd.$$

In addition, by (I60) and our assumption that $p@\{30\}$ holds, we have $q@\{3..5, 22..24\}$. We now prove that $Reset[q.node].rnd = q.y.rnd$ holds by considering the two cases

$q@\{22, 23\}$ and $q@\{3..5, 24\}$ separately. First, suppose that $q@\{22, 23\}$ holds. Then, by (I8) and (I60), we have either $Reset[q.node].rnd = q.y.rnd$ or $\mathsf{Dist}[q.y.rnd] \neq \perp$. However, because $Reset[i].rnd = q.y.rnd \wedge p@\{30\}$ holds, by (I14) and (I60), $\mathsf{Dist}[q.y.rnd]$ $= \perp$ holds. Thus, in this case, we have $Reset[q.node].rnd = q.y.rnd$.

On the other hand, if $q@\{3..5, 24\}$ holds, then by (I31) and (A.9), $A(q, q.node)$ holds as well. By (I3), this implies that $Reset[q.node].rnd = q.y.rnd$ holds.

Putting these assertions together, we have $Reset[i].rnd = Reset[q.node].rnd$. By (I15), this implies that $q.node = i$. Therefore, if $q \neq p$, we have $q@\{3..5, 22..24\} \wedge q.node = i \wedge (q@\{3\} \Rightarrow q.dir = stop)$, which implies that disjunct $\mathcal{A}$ holds both before and after the execution of $30.p$. □

The following lemma is used in proving invariants (I58) and (I59).

Lemma A.2 If t and u are consecutive states such that $p@\{2, 3, 15..24\}$ holds at t, the condition $i \xrightarrow{*} \mathsf{Loc}[p]$ holds at u but not at t, and all the invariants in this appendix hold at t, then the following are true:

- The value of $\mathsf{Loc}[p]$ is changed by a call to $\mathsf{UpdateLoc}(P, i)$ such that $p \in P$.
- $i = \mathsf{Loc}[p] \wedge C(\lfloor i/2 \rfloor) > C(i)$ holds at u.

Proof: The only statements that may establish $i \xrightarrow{*} \mathsf{Loc}[p]$ while $p@\{2, 3, 15..24\}$ holds are $16.p$, $19.p$, $20.p$, $15.q$, $27.q$, and $29.q$, where q is any arbitrary process. It should be obvious that these statements can update $\mathsf{Loc}[p]$ only by calling $\mathsf{UpdateLoc}(P, i)$, where $p \in P$. Now we show that if state u is reached via the execution of any of these statements, then $i = \mathsf{Loc}[p]$ is established.

By using (I50) and (I51), we can tabulate all the possible ways in which $\mathsf{Loc}[p]$ can be changed by one of these statements. Such a tabulation is given below. Note that the table only shows the ways in which $\mathsf{Loc}[p]$ may *change* value. For example, by (I50), if $p@\{16\}$ holds, then $\mathsf{Loc}[p]$ is either $p.node$ or $2 \cdot p.node$. In both cases, statement $16.p$ may change the value of $\mathsf{Loc}[p]$ only by establishing $\mathsf{Loc}[p] = 2 \cdot p.node + 1$. As another example, by (I51), if $p@\{19\}$ holds, then $\mathsf{Loc}[p]$ is either $p.node$ or $2 \cdot p.node$. Because $19.p$ can change the value of $\mathsf{Loc}[p]$ only by establishing $\mathsf{Loc}[p] = 2 \cdot p.node$, we do not include an entry for $\mathsf{Loc}[p] = 2 \cdot p.node$ in the column for state t.

statement	at state t	at state u
$16.p$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node + 1$
	$\mathsf{Loc}[p] = 2 \cdot p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node + 1$
$19.p$, $20.p$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node$
$15.q$, $27.q$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node$
$29.q$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node$

From this table, we see that there are only three ways in which the value of $\mathsf{Loc}[p]$ can be changed: **(i)** from a parent ($p.node$) to its left child ($2 \cdot p.node$), **(ii)** from a parent ($p.node$) to its right child ($2 \cdot p.node + 1$), and **(iii)** from a left child ($2 \cdot p.node$) to its right sibling ($2 \cdot p.node + 1$). It follows that $i \xrightarrow{*} \mathsf{Loc}[p]$ holds at u but not at t if and only if $i = \mathsf{Loc}[p]$ is established in transiting from t to u.

Our remaining proof obligation is to show that $C(\lfloor i/2 \rfloor) > C(i)$ holds at u. Let $h = \lfloor i/2 \rfloor$. Note that h is splitter i's parent. (Note further that, because $i \xrightarrow{*} \mathsf{Loc}[p]$ does not hold at t, and because the root is an ancestor of every splitter, splitter i is not the root, *i.e.*, its parent does exist.) We will establish $C(h) > C(i)$ by showing that some process "located" within the subtree rooted at h is either at h or in the subtree rooted at i's sibling. Note that in each of cases **(i)** through **(iii)**, we have

$$h = p.node \ \wedge \ (i = 2 \cdot p.node \ \vee \ i = 2 \cdot p.node + 1). \tag{A.10}$$

We now show that $C(\lfloor i/2 \rfloor) > C(i)$ holds at u by considering each of statements $16.p$, $19.p$, $20.p$, $15.q$, $27.q$, and $29.q$ separately. Statement $16.p$ may establish $\mathsf{Loc}[p] = i$, where

$$i = 2 \cdot p.node + 1 \tag{A.11}$$

only if executed when $Y[h].\mathit{free} = \mathit{false}$, in which case the antecedent of (I57) holds. Thus, one of the three disjuncts of the consequent of (I57) holds.

- If disjunct $\mathcal{A}$ holds, then by (I51)–(I53), there exists q such that $q@\{3..9, 18..38\} \wedge (\mathsf{Loc}[q + N] = h \ \vee \ \mathsf{Loc}[q + N] = 2h)$.
- If disjunct $\mathcal{B}$ holds, then by (I49)–(I53), there exists q such that $q@\{2..9, 15..40\} \wedge 2h \xrightarrow{*} \mathsf{Loc}[q + N]$.
- If disjunct $\mathcal{C}$ holds, then by (I56), there exists q such that $(q@\{9, 25\} \ \wedge \ 2h \xrightarrow{*} \mathsf{Loc}[q]) \ \vee \ (q@\{26..38\} \ \wedge \ \mathsf{Loc}[q] = q.n \ \wedge \ q.n = h)$.

If $q = p$, then the condition $p@\{16\}$ implies that disjunct $\mathcal{B}$ must hold. By (A.10), this implies that $2h = 2 \cdot p.node \xrightarrow{*} q.node$ holds, which contradicts $q = p$. Hence, $q \neq p$, and thus (by the program text) statement $16.p$ does not change the value of either $\mathsf{Loc}[q]$ or $\mathsf{Loc}[q+N]$. Because one of $\mathcal{A}$ through $\mathcal{C}$ holds before $16.p$ is executed, the following assertion holds both before and after the execution of $16.p$.

$$q \neq p \ \wedge\ \big(\mathsf{Loc}[q] = h \ \vee\ \mathsf{Loc}[q+N] = h \ \vee\ 2h \xrightarrow{*} \mathsf{Loc}[q] \ \vee\ 2h \xrightarrow{*} \mathsf{Loc}[q+N]\big)$$

In other words, when $16.p$ is executed (which moves $\mathsf{Loc}[p]$ to the subtree rooted at i), at least one of $\mathsf{Loc}[q]$ and $\mathsf{Loc}[q+N]$ is equal to h, or a splitter within the left subtree of h. Furthermore, because $16.p$ does not alter $\mathsf{Loc}[q]$ or $\mathsf{Loc}[q+N]$, this is also true after $16.p$ is executed, *i.e.*, in state u. Note that, by (A.10) and (A.11), i is the right child of h. This implies that $C(h) > C(i)$ holds at state u.

Statement $19.p$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $X[h] \neq p$ holds. However, this implies that $A(p, h)$ is false, and hence $\mathsf{Loc}[p] = 2h$, by (I51). Therefore, statement $19.p$ cannot change the value of $\mathsf{Loc}[p]$.

Statement $20.p$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $Acquired[h] = true$ holds. By (I4), this implies that there exists a process q such that $q@\{3..14, 25..41\} \wedge A(q, h)$ holds. Thus, by (I24) and (I53), $\mathsf{Loc}[q+N] = h$ holds both before and after the execution of $20.p$. This implies that $C(h) > C(i)$.

Statement $15.q$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $p@\{16..19\} \ \wedge\ p.node = h \ \wedge\ q.node = h$ holds. In this case, by (I49), $\mathsf{Loc}[q] = h$ holds at t. Because statement $15.q$ does not update $\mathsf{Loc}[q]$, this condition also holds at u, which implies that $C(h) > C(i)$.

Similarly, statements $27.q$ and $29.q$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $p@\{16..22\} \ \wedge\ p.node = h \ \wedge\ q.n = h$ holds. In this case, by (I56), $\mathsf{Loc}[q] = h$ holds at t, and hence at u. This implies that $C(h) > C(i)$. $\square$

invariant $p@\{2, 3, 15..24\} \ \wedge\ (i \xrightarrow{*} \mathsf{Loc}[p]) \ \Rightarrow\ \mathsf{PC}[p, i] \geq C(i)$ (I58)

Proof: The only statement that can establish $p@\{2,\ 3,\ 15..24\}$ is $1.p$. But this establishes $\mathsf{Loc}[p] = 1$ and $\mathsf{PC}[p,\ 1] = C(1)$. Hence the consequent is established (or preserved) for $i = 1$, and the antecedent is falsified for $i \neq 1$.

If $p@\{2, 3, 15..24\}$ holds, then the only statements that can establish $i \xrightarrow{*} \mathsf{Loc}[p]$ are $16.p$, $19.p$, $20.p$, $15.q$, $27.q$, and $29.q$, where q is any arbitrary process. By Lemma A.2, if one of these statements establishes $i \xrightarrow{*} \mathsf{Loc}[p]$, it establishes it by calling $\mathsf{UpdateLoc}(P, i)$ such that $p \in P$. In this case, line u2 of UpdateLoc establishes $\mathsf{PC}[p, i] = C(i)$.

The value of $\mathsf{PC}[p, i]$ may be changed only by statement $1.p$, or by a call to UpdateLoc. However, statement $1.p$ establishes the consequent as shown above, and whenever UpdateLoc updates $\mathsf{PC}[p, i]$, it establishes $\mathsf{PC}[p, i] = C(i)$.

The value of $C(i)$ may be changed only by statements $14.q$, $25.q$, and $41.q$ (by updating $\mathsf{Loc}[q]$ or $\mathsf{Loc}[q + N]$ directly), or by a call to UpdateLoc. However, by (I55), statement $25.q$ always changes $\mathsf{Loc}[q]$ from a splitter to its ancestor. It follows that statement $25.q$ cannot cause $C(i)$ to increase for any i. Similarly, statements $14.q$ and $41.q$ establishes $\mathsf{Loc}[q] = 0 \ \wedge\ \mathsf{Loc}[q + N] = 0$, and hence they also cannot cause $C(i)$ to increase for any i.

The only remaining case is when $C(i)$ is changed by a call to UpdateLoc. However, line u3 of UpdateLoc ensures that (I58) is always preserved in this case. □

invariant $p@\{2, 3, 15..24\} \ \wedge\ (i \xrightarrow{*} \mathsf{Loc}[p]) \ \wedge\ (2 \le i \le T) \ \Rightarrow$
$$\mathsf{PC}[p, i] < \mathsf{PC}[p, \lfloor i/2 \rfloor] \qquad \text{(I59)}$$

Proof: The only statement that can establish $p@\{2, 3, 15..24\}$ is $1.p$, but this establishes $\mathsf{Loc}[p] = 1$, and hence falsifies the antecedent.

The only other statements that may establish the antecedent (by changing the value of $\mathsf{Loc}[p]$) are $14.p$, $16.p$, $19.p$, $20.p$, $25.p$, $41.p$, $15.q$, $27.q$, and $29.q$, where q is any arbitrary process. Statements $14.p$, $25.p$, and $41.p$, falsify the antecedent. The other statements may establish the antecedent only by calling UpdateLoc when $p@\{2, 3, 15..24\}$ holds. Similarly, the only statements that may falsify the consequent (by changing the value of $\mathsf{PC}[p, i]$ or $\mathsf{PC}[p, \lfloor i/2 \rfloor]$) are $1.q$, $15.q$, $16.q$, $19.q$, $20.q$, $27.q$, and $29.q$, where q is any arbitrary process. These statements also may falsify (I59) only by calling UpdateLoc when $p@\{2, 3, 15..24\}$ holds. Therefore, it suffices to consider such an invocation of UpdateLoc. We consider two cases. Let h be the parent of i, *i.e.*, $h = \lfloor i/2 \rfloor$.

Case 1: $i \xrightarrow{*} \mathsf{Loc}[p]$ is established by $\mathsf{UpdateLoc}(P, f)$.

By Lemma A.2, $i \xrightarrow{*} \mathsf{Loc}[p]$ can be established only if $p \in P \ \wedge\ f = i$, and in this case $i = \mathsf{Loc}[p] \ \wedge\ C(\lfloor i/2 \rfloor) > C(i)$ is also established. Note that line u2 of UpdateLoc establishes $\mathsf{PC}[p, i] = C(i)$. Also, because $p@\{2, 3, 15..24\}$ holds, line u3 ensures that $\mathsf{PC}[p, h] \ge C(h)$ holds after the call to UpdateLoc. Thus, $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$ holds after UpdateLoc is called.

Case 2: $i \xrightarrow{*} \mathsf{Loc}[p]$ holds before the call to $\mathsf{UpdateLoc}(P, f)$.

In this case, the antecedent of (I59) holds before the call to UpdateLoc, and hence $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$ holds as well. If the value of $\mathsf{PC}[p, h]$ is changed by line u2 of UpdateLoc, then $p \in P$ and $f = h$. But this implies that line u1 establishes $\mathsf{Loc}[p] = h$, which falsifies the antecedent of (I59). (In fact, such a case can never occur, because it implies that a process moves *upward* within the renaming tree while in its entry section.) If the value of $\mathsf{PC}[p, h]$ is changed by line u3, then because line u3 never causes a PC entry to decrease, the condition $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$ cannot be falsified.

The remaining possibility to consider is that the value of $\mathsf{PC}[p, i]$ is changed when $\mathsf{UpdateLoc}(P, f)$ is called. Note that, if $\mathsf{PC}[p, i]$ is changed, either by line u2 or line u3, then

$$\mathsf{PC}[p, i] = C(i) \tag{A.12}$$

is established. From (I58), it follows that the value of $\mathsf{PC}[p, i]$ can increase only if $C(i)$ also increases. By the definition of $C(i)$, this can happen only if $\mathsf{UpdateLoc}(P, f)$ establishes either $i \xrightarrow{*} \mathsf{Loc}[r]$ or $i \xrightarrow{*} \mathsf{Loc}[r + N]$ for some $r \in P$. In this case, either $\mathsf{UpdateLoc}(P, f)$ is called by statement $1.r$, or $r@\{2, 3, 15..24\}$ holds before the execution of the statement calling $\mathsf{UpdateLoc}(P, f)$ (this can be seen by examining each call to UpdateLoc in Figure A.1). However, statement $1.r$ calls $\mathsf{UpdateLoc}(\{r\}, 1)$, and hence cannot establish $i \xrightarrow{*} \mathsf{Loc}[r]$ for any $i \geq 2$. Therefore, $r@\{2, 3, 15..24\}$ holds before UpdateLoc is called. By (I49)–(I52), this implies that $\mathsf{Loc}[r] = \mathsf{Loc}[r + N]$ also holds. Therefore, it is enough to consider the case in which $i \xrightarrow{*} \mathsf{Loc}[r]$ is established.

By Lemma A.2, $\mathsf{UpdateLoc}(P, f)$ establishes $i \xrightarrow{*} \mathsf{Loc}[r]$ only if it also establishes $C(\lfloor i/2 \rfloor) > C(i)$, *i.e.*, $C(h) > C(i)$. As noted earlier, line u3 ensures that $\mathsf{PC}[p, h] \geq C(h)$ holds after the call to UpdateLoc. Thus, by (A.12), we again have $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$. □

invariant (Adaptivity) $p@\{4..8\} \Rightarrow \mathsf{lev}(p.node) < \mathsf{PC}[p, 1]$ (I61)

Proof: The antecedent is established only by statement $3.p$. We begin by showing that if $3.p$ establishes the antecedent by establishing $p@\{4, 6\}$, then $\mathsf{PC}[p, 1] \geq \mathsf{PC}[p, p.node] + p.level$ holds after its execution. Note that $3.p$ establishes $p@\{4, 6\}$ only if executed when $p.dir = stop \lor p.level \geq L$ holds. If $3.p$ is executed when $p.level = 0 \land p.dir = stop$ holds, then by (I35), $p.node = 1$ holds as well. $3.p$ does not update either $p.node$ or $p.level$ in this case, and hence, $\mathsf{PC}[p, 1] \geq \mathsf{PC}[p, p.node] + p.level$ holds after its execution.

The remaining possibility to consider is that $(p.dir = stop \wedge p.level > 0) \vee p.level \geq L$ holds before $3.p$ is executed. In this case, by (I46), $3.p$ establishes the following.

$$\begin{array}{l} p.path[0].node = 1 \;\wedge\; p.path[p.level-1].node \xrightarrow{*} p.node \quad \wedge \\ (\forall l : 0 \leq l < p.level - 1 :: p.path[l].node \xrightarrow{*} p.path[l+1].node). \end{array}$$

By (I59), the following is established as well.

$$\begin{array}{rcl} \mathsf{PC}[p,1] & = & \mathsf{PC}[p, p.path[0].node] \\ & > & \mathsf{PC}[p, p.path[1].node] \\ & & \quad \vdots \\ & > & \mathsf{PC}[p, p.path[p.level-1].node] \\ & > & \mathsf{PC}[p, p.node]. \end{array}$$

Given the length of this sequence, we have $\mathsf{PC}[p,1] \geq \mathsf{PC}[p, p.node] + p.level$, as claimed.

By (I35), this implies that $\mathsf{PC}[p,1] \geq \mathsf{PC}[p, p.node] + \mathsf{lev}(p.node)$ holds after $3.p$ is executed. Note that $p@\{4, 6\}$ implies that $\mathsf{PC}[p, p.node] \geq 1$ holds (the point contention for some process at a splitter must at least include that process). Hence, if $3.p$ establishes the antecedent of (I61), then $\mathsf{PC}[p,1] > \mathsf{lev}(p.node)$ holds after its execution.

While the antecedent holds, the value of $\mathsf{lev}(p.node)$ cannot be changed. Moreover, if $\mathsf{PC}[p,1]$ is changed within $\mathsf{UpdateLoc}$, then its value is increased. $\square$

Note that $\mathsf{PC}[p,1]$ is initialized by p at statement $1.p$ to match the current contention within the renaming tree, assuming that each process q is counted twice, as "q" and as "$q+N$." While $p@\{2, 3, 15..24\}$ continues to hold, as other processes enter and leave the renaming tree, if the current contention ever exceeds the current the value of $\mathsf{PC}[p,1]$, then line $\mathsf{u3}$ of $\mathsf{UpdateLoc}$ ensures that $\mathsf{PC}[p,1]$ is updated accordingly. Thus, if m is the maximum value attained by $\mathsf{PC}[p,1]$ while $p@\{2,3,15..24\}$ holds, then m is at most twice the actual point contention experienced by p in its entry section.

It should be clear that the number of remote memory references executed by p to enter and then exit its critical section is $\Theta(n)$, where n is the value of $\mathsf{lev}(p.node)$ when p reaches statement 4 or 6. By (I61), we have $n < m$. Clearly, we also have $n \leq L+1$. Thus, p executes $O(\min(k, \log N))$ remote memory references to enter and then exit its critical section, where k is the point contention it experiences in its entry section.

APPENDIX B

DETAILED PROOFS OF LEMMAS 5.1–5.6

In this appendix, full proofs are presented for Lemmas 5.1–5.6. Throughout this appendix, we use the definitions stated in Section 5.1. As in Chapter 5, we omit *RFS* when quoting properties RF1–RF5 and assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$.

Lemma 5.1 Consider a computation H and two sets of processes *RFS* and Y. Assume the following:

- $H \in C$; (B.1)
- *RFS* is a valid RF-set of H; (B.2)
- $\mathit{RFS} \subseteq Y$. (B.3)

Then, the following hold: $H \mid Y \in C$; *RFS* is a valid RF-set of $H \mid Y$; an event e in $H \mid Y$ is a critical event if and only if it is also a critical event in H.

Proof: By (B.2), H satisfies RF1–RF5. Since H satisfies RF1, if a process p is not in *RFS*, no process other than p reads a value written by p. Therefore, by inductively applying P2, we have $H \mid Y \in C$.

We now prove that *RFS* is a valid RF-set of $H \mid Y$.

- **RF1:** Assume that H can be written as $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$, and that $H \mid Y$ can be written as $(E \mid Y) \circ \langle e_p \rangle \circ (F \mid Y) \circ \langle f_q \rangle \circ (G \mid Y)$. Also assume that $p \neq q$ and that there exists a variable $v \in \mathit{Wvar}(e_p) \cap \mathit{Rvar}(f_q)$ such that $F \mid Y$ does not contain a write to v. We claim that F does not contain a write to v, in which case, by applying RF1 to H, we have $p \in \mathit{RFS}$.

  Assume, to the contrary, that F has a write to v. Define $g_r = \mathit{writer\_event}(v, F)$. Since $F \mid Y$ does not contain g_r, we have $r \notin Y$. Since $H \mid Y$ contains f_q, we have $q \in Y$. Therefore, $r \neq q$. By applying RF1 to g_r and f_q in H, we have $r \in \mathit{RFS}$, which contradicts $r \notin Y$.

- **RF2:** Consider an event e_p in $H \mid Y$, and a variable v in $var(e_p)$. Assume that v is local to another process q $(\neq p)$. By applying RF2 to H, we have either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq RFS$. Since $\mathrm{Act}(H \mid Y) \subseteq \mathrm{Act}(H)$, $q \notin \mathrm{Act}(H)$ implies $q \notin \mathrm{Act}(H \mid Y)$.

- **RF3:** Consider a variable $v \in V$ and two different events e_p and f_q in $H \mid Y$. Assume that both p and q are in $\mathrm{Act}(H \mid Y)$, $p \neq q$, there exists a variable v such that $v \in var(e_p) \cap var(f_q)$, and there exists a write to v in $H \mid Y$. We claim that $writer(v, H \mid Y) \in RFS$ holds.

  By definition, events e_p and f_q also exist in H, and there exists a write to v in H. Since $\mathrm{Act}(H \mid Y) \subseteq \mathrm{Act}(H)$, both p and q are in $\mathrm{Act}(H)$. Therefore, by applying RF3 to H, we have $writer(v, H) \in RFS$. Thus, since $RFS \subseteq Y$, the last event to write to v is identical in both H and $H \mid Y$. Hence, $writer(v, H \mid Y) \in RFS$ holds.

- **RF4:** From $\mathrm{Act}(H \mid Y) \subseteq \mathrm{Act}(H)$, $\mathrm{Fin}(H \mid Y) \subseteq \mathrm{Fin}(H)$, and the fact that H satisfies RF4, it easily follows that $H \mid Y$ satisfies RF4.

- **RF5:** If an event e_p in $H \mid Y$ is critical in $H \mid Y$, then by the definition of a critical event, it is also critical in H. (Note that every event that is contained in H but not in $H \mid Y$ is executed by some process different from p. Adding such events to $H \mid Y$ cannot make e_p noncritical.) Thus, by applying RF5 to H, e_p is also critical in $H \mid (\{p\} \cup RFS)$.

Finally, we claim that an event e_p in $H \mid Y$ is a critical event if and only if it is also a critical event in H. If e_p is critical in $H \mid Y$, then as shown above (in the reasoning for RF5), it is also critical in H. On the other hand, if e_p is critical in H, then by applying RF5 to H, it is also critical in $H \mid (\{p\} \cup RFS)$. Since e_p is an event of $H \mid Y$, we have $p \in Y$, and hence $\{p\} \cup RFS \subseteq Y$. Thus, by the definition of a critical event, e_p is also critical in $H \mid Y$. □

Lemma 5.2 Consider three computations H, H', and G, a set of processes RFS, and two events e_p and e'_p of a process p. Assume the following:

- $H \circ \langle e_p \rangle \in C$; (B.4)
- $H' \circ G \circ \langle e'_p \rangle \in C$; (B.5)
- RFS is a valid RF-set of H; (B.6)

- RFS is a valid RF-set of H'; (B.7)
- $e_p \sim e'_p$; (B.8)
- $p \in \mathrm{Act}(H)$; (B.9)
- $H \mid (\{p\} \cup RFS) = H' \mid (\{p\} \cup RFS)$; (B.10)
- $G \mid p = \langle\rangle$; (B.11)
- no events in G write any of p's local variables; (B.12)
- e_p is critical in $H \circ \langle e_p \rangle$. (B.13)

Then, e'_p is critical in $H' \circ G \circ \langle e'_p \rangle$. Moreover, if the following conditions are true,

(A) $H' \circ G$ satisfies RF5;

(B) if e_p is a comparison event on a variable v, and if G contains a write to v, then $G \mid RFS$ also contains a write to v.

then $H' \circ G \circ \langle e'_p \rangle$ also satisfies RF5.

Proof: First, define

$$\overline{H} = H' \circ G \circ \langle e'_p \rangle. \tag{B.14}$$

Note that, by (B.9), (B.10) and (B.11), we have the following:

$$\begin{aligned} H \mid p &= (H' \circ G) \mid p, \quad \text{and} && \text{(B.15)} \\ p &\in \mathrm{Act}(H'). && \text{(B.16)} \end{aligned}$$

If Condition (A) is true, then in order to show that $\overline{H}$ satisfies RF5, it suffices to consider event e'_p. If e'_p is a critical read or transition event, then it is clearly critical in $\overline{H} \mid (\{p\} \cup RFS)$. Thus, our remaining proof obligations are to show

- e'_p is critical in $\overline{H}$;
- if e'_p is a critical write or a critical comparison, and if Condition (B) is true, then e'_p is also a critical write or a critical comparison in $\overline{H} \mid (\{p\} \cup RFS)$.

We consider four cases, depending on the kind of critical event e_p is.

Transition event: If e_p is a transition event, then by (B.8) and the definition of congruence, e'_p is also a transition event.

Critical read: If e_p is a critical read in $H \circ \langle e_p \rangle$, then there exists a variable v, remote to p, such that $op(e_p) = \mathsf{read}(v)$ and $H \mid p$ does not contain a read from v. Thus, by (B.8) and (B.15), e'_p is also a critical read in $\overline{H}$.

Before considering the remaining two cases, note that, if e_p is a critical write or a critical comparison in $H \circ \langle e_p \rangle$, then there exists a variable v, remote to p, such that $v \in var(e_p)$ and

$$writer(v, H) \neq p. \tag{B.17}$$

Critical write: Assume that e_p is a critical write in $H \circ \langle e_p \rangle$. We consider two cases. First, if p does not write v in H, then by (B.15),

- p does not write v in $H' \circ G$. (B.18)

Thus, we have $writer(v, H' \circ G) \neq p$, and hence, by (B.8) and the definition of a critical write, e'_p is a critical write in $\overline{H}$. Moreover, by (B.18), we also have $writer(v, (H' \circ G) \mid (\{p\} \cup RFS)) \neq p$. Thus, e'_p is also a critical write in $\overline{H} \mid (\{p\} \cup RFS)$.

Second, if p writes to v in H, then define $\bar{e}_p$ to be the last event by p that writes to v in H. Define $q = writer(v, H)$ and $f_q = writer\_event(v, H)$. By (B.17), we have $q \neq p$ and $q \neq \perp$. If $q \in \mathrm{Act}(H)$, then by (B.6) and (B.9), and applying RF3 to $\bar{e}_p$ and f_q in H, we have $q \in RFS$. On the other hand, if $q \in \mathrm{Fin}(H)$, then clearly $q \in RFS$ by (B.6). Thus, in either case, in H, there exists a write to v (that is, f_q) by some process in RFS after the last write to v by p (that is, $\bar{e}_p$). Therefore, by (B.10), the same is true in H', and hence by (B.11), we have $writer(v, H' \circ G) \neq p$. It follows, by (B.8) and the definition of a critical write, that e'_p is a critical write in $\overline{H}$.

Moreover, since $q \in RFS$, there also exists a write to v (that is, f_q) after the last write to v by p ($\bar{e}_p$) in $\overline{H} \mid (\{p\} \cup RFS)$. It follows that e'_p is also a critical write in $\overline{H} \mid (\{p\} \cup RFS)$.

Critical comparison: Assume that e_p is a critical comparison on v in $H \circ \langle e_p \rangle$. We consider three cases.

Case 1: If G contains a write to v, then by (B.11) and the definition of a critical comparison, e'_p is clearly a critical comparison in $\overline{H}$. Moreover, if (B) is true, then since G contains a write to v, $G \mid RFS$ also contains a write to v, and hence e'_p is also a critical comparison in $\overline{H} \mid (\{p\} \cup RFS)$.

Case 2: If p does not access v in H (*i.e.*, for every event f_p in $H \mid p$, $v \notin var(f_p)$), then by (B.15), p does not access v in $H' \circ G$. Thus, by (B.8) and the definition of a critical comparison, e'_p is a critical comparison in $\overline{H}$. Moreover, it is clear that p does not access v in $(H' \circ G) \mid (\{p\} \cup RFS)$. Thus, e'_p is also a critical comparison in $\overline{H} \mid (\{p\} \cup RFS)$.

Case 3: Assume that p accesses v in H, and that G does not contain a write to v. Then, there exists an event f_p in H such that $v \in var(f_p)$. Moreover, since v is the only remote variable in $var(e_p)$, by (B.12), it follows that

- for each variable u in $Rvar(e_p)$, G does not contain a write to u. (B.19)

We now establish two claims.

Claim 1: $writer(v, H) = writer(v, H')$ and $writer\_event(v, H) = writer\_event(v, H')$. Moreover, either both $writer(v, H)$ and $writer(v, H')$ are $\perp$, or both are in $\{p\} \cup RFS$.

Proof of Claim: Suppose that v is written in H and let g_{q_v} be the last event to do so. If $q_v \neq p$ and if $q_v \in \mathrm{Act}(H)$, then by (B.6) and (B.9), and by applying RF3 to f_p and g_{q_v} in H, we have $q_v \in RFS$. On the other hand, if $q_v \in \mathrm{Fin}(H)$, then clearly $q_v \in RFS$ holds by (B.6). Finally, if $q_v = p$, then clearly we have $q_v \in \{p\} \cup RFS$. Thus, in all cases, we have $q_v \in \{p\} \cup RFS$. Similarly, if $g_{q'_v}$ is the last event to write v in H', then by (B.7) and (B.16), we have $q'_v \in \{p\} \cup RFS$. Therefore, by (B.10), it follows that the last event to write v (if any) is the same in H and H'. Hence, the claim follows. $\square$

Claim 2: For each variable u in $Rvar(e_p)$, $writer\_event(u, H) = writer\_event(u, H')$ holds.

Proof of Claim: If u is remote to p, then by the Atomicity property, we have $u = v$. Thus, in this case, the claim follows by Claim 1.

So, assume that u is local to p. If there exists an event g_{q_u} in H that writes u, then by (B.6) and (B.9), and by applying RF2 to g_{q_u} in H, it follows that either $q_u = p$ or $q_u \in RFS$ holds. Similarly, if an event $g_{q'_u}$ writes u in H', then by (B.7) and (B.16), we have either $q'_u = p$ or $q'_u \in RFS$. Therefore, by (B.10), it follows that the last event to write u (if any) is the same in H and H'. $\square$

From (B.19) and Claim 2, it follows that

- for each variable u in $\mathit{Rvar}(e_p)$, $\mathit{value}(u, H) = \mathit{value}(u, H' \circ G)$ holds. (B.20)

By (B.4), (B.15), (B.20), and P2, we have $H' \circ G \circ \langle e_p \rangle \in C$. Combining with (B.5), and using P5, we have $e_p = e'_p$. In particular,

- e'_p is a successful (respectively, unsuccessful) comparison in $\overline{H}$ if and only if e_p is also a successful (respectively, unsuccessful) comparison in $H \circ \langle e_p \rangle$. (B.21)

Define $q = \mathit{writer}(v, H)$ and $g_q = \mathit{writer\_event}(v, H)$. By (B.17) and Claim 1, we have

$$q = \bot \quad \vee \quad q \in \mathit{RFS} - \{p\}, \tag{B.22}$$

and $g_q = \mathit{writer\_event}(v, H')$. Thus, by (B.10),

$$q \neq \bot \;\Rightarrow\; H \mid \{p, q\} = H' \mid \{p, q\}. \tag{B.23}$$

Since G does not contain a write to v, we also have

$$\mathit{writer}(v, H' \circ G) = q \qquad \wedge \qquad \mathit{writer\_event}(v, H' \circ G) = g_q. \tag{B.24}$$

If e_p is a critical successful comparison in $H \circ \langle e_p \rangle$, then by (B.21), (B.22), (B.24), and the definition of a critical successful comparison, e'_p is clearly a critical successful comparison in both $\overline{H}$ and $\overline{H} \mid (\{p\} \cup \mathit{RFS})$.

Similarly, if e_p is a critical unsuccessful comparison in $H \circ \langle e_p \rangle$, then by definition, either $H \mid p$ does not contain an unsuccessful comparison event by p on v, or $q \neq \bot$ and H does not contain an unsuccessful comparison event by p on v *after* g_q. In the former case, by (B.15), $(H' \circ G) \mid p$ does not contain an unsuccessful comparison event on v; in the latter case, by (B.11) and (B.23), $H' \circ G$ does not contain an unsuccessful comparison event by p on v *after* g_q.

Therefore, In either case, by (B.21), it follows that e'_p is a critical unsuccessful comparison in both $\overline{H}$ and $\overline{H} \mid (\{p\} \cup \mathit{RFS})$. $\square$

Lemma 5.3 Consider two computations H and G, a set of processes RFS, and an event e_p of a process p. Assume the following:

- $H \circ G \circ \langle e_p \rangle \in C$; (B.25)
- RFS is a valid RF-set of H; (B.26)
- $p \in \mathrm{Act}(H)$; (B.27)

- $H \circ G$ satisfies RF1 and RF2; (B.28)
- G is an Act(H)-computation; (B.29)
- $G \,|\, p = \langle\rangle$; (B.30)
- if e_p remotely accesses a variable v_{rem}, then the following hold:
  - if v_{rem} is local to a process q, then either $q \notin \text{Act}(H)$ or $\{p, q\} \subseteq \mathit{RFS}$, and (B.31)
  - if $q = \mathit{writer}(v_{\text{rem}}, H \circ G)$, then one of the following hold: $q = \perp$, $q = p$, $q \in \mathit{RFS}$, or $v_{\text{rem}} \notin \mathit{Rvar}(e_p)$. (B.32)

Then, $H \circ G \circ \langle e_p \rangle$ satisfies RF1 and RF2.

Proof: Define
$$\overline{H} = H \circ G \circ \langle e_p \rangle. \tag{B.33}$$

By (B.27) and (B.30), we have
$$p \in \text{Act}(H \circ G). \tag{B.34}$$

By (B.29), we also have $\text{Act}(H \circ G) \subseteq \text{Act}(H)$. Thus, by (B.34),
$$\text{Act}(\overline{H}) \subseteq \text{Act}(H \circ G) \subseteq \text{Act}(H). \tag{B.35}$$

Now we prove each of RF1 and RF2 separately.

- **RF1:** Since, by (B.28), $H \circ G$ satisfies RF1, it suffices to consider the following case: $\overline{H}$ can be written as $E \circ \langle f_q \rangle \circ F \circ \langle e_p \rangle$; $p \neq q$; there exists a variable $v \in \mathit{Wvar}(f_q) \cap \mathit{Rvar}(e_p)$; and F does not contain a write to v. Our proof obligation is to show $q \in \mathit{RFS}$.

  If v is local to p, then by (B.28), and applying RF2 to f_q in $H \circ G$, we have either $p \notin \text{Act}(H \circ G)$ or $q \in \mathit{RFS}$. Thus, by (B.34), we have $q \in \mathit{RFS}$. On the other hand, if v is remote to p, then we have $v_{\text{rem}} = v$, which implies $q = \mathit{writer}(v_{\text{rem}}, H \circ G)$, $q \neq \perp$, $q \neq p$, and $v_{\text{rem}} \in \mathit{Rvar}(e_p)$. Thus, by (B.32), we have $q \in \mathit{RFS}$.

- **RF2:** Consider an event f_q in $\overline{H}$, and a variable v in $\mathit{var}(f_q)$. Assume that v is local to another process $r \neq q$. Our proof obligation is to show that either $r \notin \text{Act}(\overline{H})$ or $\{q, r\} \subseteq \mathit{RFS}$ holds.

If f_q is an event of $H \circ G$, then by (B.28), and applying RF2 to f_q in $H \circ G$, we have either $r \notin \mathrm{Act}(H \circ G)$ or $\{q, r\} \subseteq \mathit{RFS}$. By (B.35), $r \notin \mathrm{Act}(H \circ G)$ implies $r \notin \mathrm{Act}(\overline{H})$.

On the other hand, if $f_q = e_p$, then we have $p = q$ and $v_{\mathrm{rem}} = v$. By applying (B.31) with 'q' $\leftarrow r$, we have either $r \notin \mathrm{Act}(H)$ or $\{p, r\} = \{q, r\} \subseteq \mathit{RFS}$. By (B.35), $r \notin \mathrm{Act}(H)$ implies $r \notin \mathrm{Act}(\overline{H})$. $\square$

In order to prove Lemma 5.4, we need several more lemmas, presented here. The next lemma shows that appending a noncritical event of an active process does not invalidate a valid RF-set.

Lemma B.1 Consider a computation H, a set of processes RFS, and an event e_p of a process p. Assume the following:

- $H \circ \langle e_p \rangle \in C$; (B.36)
- RFS is a valid RF-set of H; (B.37)
- $p \in \mathrm{Act}(H)$; (B.38)
- e_p is noncritical in $H \circ \langle e_p \rangle$. (B.39)

Then, RFS is a valid RF-set of $H \circ \langle e_p \rangle$.

Proof: First, note that $\mathrm{Act}(H) = \mathrm{Act}(H \circ \langle e_p \rangle)$ and $\mathrm{Fin}(H) = \mathrm{Fin}(H \circ \langle e_p \rangle)$, because $p \in \mathrm{Act}(H)$ and $e_p \neq \mathit{Exit}_p$. If e_p remotely accesses a remote variable, then we will denote that variable as v_{rem}. By (B.39) and the definition of a critical event,

- if e_p remotely accesses v_{rem}, then there exists an event $\bar{e}_p$ in H that remotely accesses v_{rem}. (B.40)

Thus, by (B.37), and applying RF2 to $\bar{e}_p$ in H, it follows that

- if e_p remotely accesses v_{rem} and v_{rem} is local to another process q, then either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq \mathit{RFS}$ holds. (B.41)

Define $z = \mathit{writer}(v_{\mathrm{rem}}, H)$ and $f_z = \mathit{writer\_event}(v_{\mathrm{rem}}, H)$. We claim that one of the following holds: $z = \perp$, $z = p$, or $z \in \mathit{RFS}$. Assume, to the contrary, that $z \neq \perp$, $z \neq p$, and $z \notin \mathit{RFS}$. Then, by (B.37), $z \notin \mathit{RFS}$ implies $z \notin \mathrm{Fin}(H)$. Since $H \mid z \neq \langle\rangle$, we have $z \in \mathrm{Act}(H)$. Thus, by (B.38), and applying RF3 to $\bar{e}_p$ and f_z in H, we have $z \in \mathit{RFS}$, a contradiction. Thus, we have shown that

- if e_p remotely accesses v_{rem}, and if $z = \mathit{writer}(v_{\mathrm{rem}}, H)$, then one of the following hold: $z = \bot$, $z = p$, or $z \in \mathit{RFS}$. (B.42)

We now consider each condition RF1–RF5 separately.

- **RF1 and RF2:** Define $G = \langle\rangle$. It follows trivially from (B.37) that

  - $H \circ G$ satisfies RF1 and RF2. (B.43)

  We now apply Lemma 5.3. Assumptions (B.25)–(B.32) stated in Lemma 5.3 follow from (B.36), (B.37), (B.38), (B.43), $G = \langle\rangle$, $G = \langle\rangle$, (B.41), and (B.42), respectively. It follows that $H \circ \langle e_p\rangle$ satisfies RF1 and RF2.

- **RF3:** Consider a variable $v \in V$ and two different events f_q and g_r in $H \circ \langle e_p\rangle$. Assume that both q and r are in $\mathrm{Act}(H) = \mathrm{Act}(H \circ \langle e_p\rangle)$, $q \neq r$, that there exists a variable v such that $v \in \mathit{var}(f_q) \cap \mathit{var}(g_r)$, and that there exists a write to v in $H \circ \langle e_p\rangle$. Let

  $$s = \mathit{writer}(v, H \circ \langle e_p\rangle). \tag{B.44}$$

  Our proof obligation is to show that $s \in \mathit{RFS}$ holds.

  – First, assume that v is local to p. Since at least one of q or r is different from p, by (B.37) and (B.38), and applying RF2 to H, we have $p \in \mathit{RFS}$. If $s = p$, then we have $s \in \mathit{RFS}$. On the other hand, if $s \neq p$, then (B.44) implies that e_p does not write v. Hence, we have $s = \mathit{writer}(v, H)$. Therefore, by (B.37) and (B.38), and applying RF2 to $\mathit{writer\_event}(v, H)$, we have $s \in \mathit{RFS}$.

  – Second, assume that v is remote to p, and consider the case in which both f_q and g_r are in H. If $v \in \mathit{Wvar}(e_p)$, then by (B.39) and (B.44), we have $\mathit{writer}(v, H) = s = p$. (Otherwise, e_p would be either a critical write or a critical successful comparison by definition.) On the other hand, if $v \notin \mathit{Wvar}(e_p)$, then clearly we have $\mathit{writer}(v, H) = s$. Therefore, in either case, there is a write to v in H. Therefore, by (B.37), and applying RF3 to f_q and g_r in H, we have $\mathit{writer}(v, H) \in \mathit{RFS}$, and hence $s \in \mathit{RFS}$.

  – Third, assume that v is remote to p, and consider the case in which one of f_q or g_r is e_p. Without loss of generality, we can assume that f_q is in H, $g_r = e_p$, $v = v_{\mathrm{rem}}$, and $q \neq p$. We consider two cases.

  If e_p writes v_{rem}, then by (B.44), we have $s = p$. Moreover, by (B.39), we have $\mathit{writer}(v_{\mathrm{rem}}, H) = p$. (Otherwise, e_p would be either a critical write or a critical successful comparison by definition.) Applying RF3 to $\mathit{writer\_event}(v, H)$ (by p) and f_q in H, we have $\mathit{writer}(v_{\text{rem}}, H) \in \mathit{RFS}$, which implies $s \in \mathit{RFS}$.

Otherwise, if e_p does not write v_{rem}, then since $v_{\text{rem}} \in \mathit{var}(e_p)$, we have $v_{\text{rem}} \in \mathit{Rvar}(e_p)$. By (B.37), (B.38), and (B.40), and applying RF3 to $\bar{e}_p$ and f_q, we have $\mathit{writer}(v_{\text{rem}}, H) \in \mathit{RFS}$. Since e_p does not write to v_{rem}, we have $s = \mathit{writer}(v_{\text{rem}}, H) \in \mathit{RFS}$.

- **RF4:** Since H satisfies RF4, and since e_p is not one of Enter_p, CS_p, or Exit_p, it easily follows that $H \circ \langle e_p \rangle$ also satisfies RF4.

- **RF5:** This condition follows trivially from (B.37) and (B.39). □

Corollary B.1 Consider a computation H, a set of processes RFS, and another computation L. Assume the following:

- $H \circ L \in C$; (B.45)
- RFS is a valid RF-set of H; (B.46)
- L is an $\mathrm{Act}(H)$-computation; (B.47)
- L has no critical events in $H \circ L$. (B.48)

Then, RFS is a valid RF-set of $H \circ L$.

Proof: The proof of Corollary B.1 easily follows by induction on the length of L, applying Lemma B.1 at each induction step. □

The following lemma shows that if two computations H and H' are "similar enough" with respect to a process p, and if a noncritical event e_p can be appended to H, then it can also be appended to H' without modification.

Lemma B.2 Consider two computations H and H', a set of processes RFS, and an event e_p of a process p. Assume the following:

- $H \circ \langle e_p \rangle \in C$; (B.49)
- $H' \in C$; (B.50)
- RFS is a valid RF-set of H; (B.51)
- RFS is a valid RF-set of H'; (B.52)
- $p \in \mathrm{Act}(H)$; (B.53)
- $H \mid (\{p\} \cup \mathit{RFS}) = H' \mid (\{p\} \cup \mathit{RFS})$; (B.54)

- e_p is noncritical in $H \circ \langle e_p \rangle$. (B.55)

Then, the following hold: $H' \circ \langle e_p \rangle \in C$; *RFS* is a valid RF-set of both $H \circ \langle e_p \rangle$ and $H' \circ \langle e_p \rangle$; e_p is a noncritical event in $H' \circ \langle e_p \rangle$.

Proof: By (B.53) and (B.54), we have

$$p \in \mathrm{Act}(H'). \tag{B.56}$$

First, we prove that $H' \circ \langle e_p \rangle \in C$ holds. Because $H \mid p = H' \mid p$, by P2, it suffices to show that for each variable v in $Rvar(e_p)$, $writer\_event(v, H) = writer\_event(v, H')$.

- If v is local to p, then by (B.51), (B.52), and (B.53), for any event f_q in either H or H', by applying RF2 to f_q, $v \in var(f_q)$ implies $q \in \{p\} \cup RFS$. Thus, by (B.54), the last event to write to v is identical in both H and H'.

- If v is remote to p, then by (B.55) and by the definition of a critical event, there exists an event $\bar{e}_p$ by p in H such that $v \in var(\bar{e}_p)$. We consider two cases.

  First, assume that there exists a write to v in H. Define $q = writer(v, H)$ and $f_q = writer\_event(v, H)$. We claim that either $q = p$ or $q \in RFS$. If $q \in \mathrm{Fin}(H)$, then by (B.51), $q \in RFS$ follows. On the other hand, if $q \in \mathrm{Act}(H)$ and $q \neq p$ hold, then by (B.51) and (B.53), and applying RF3 to $\bar{e}_p$ and f_q in H, we have $q \in RFS$.

  Similarly, if there exists a write to v in H', then define $q' = writer(v, H')$. Since $H \mid (\{p\} \cup RFS) = H' \mid (\{p\} \cup RFS)$, H also contains a write to v if and only if H' contains a write to v, and the last event to write to v is identical in H and H'.

Thus, we have

$$H' \circ \langle e_p \rangle \in C. \tag{B.57}$$

By Lemma B.1, *RFS* is a valid RF-set of $H \circ \langle e_p \rangle$, which establishes our second proof obligation. Assumptions (B.36)–(B.39) stated in Lemma B.1 follow from (B.49), (B.51), (B.53), and (B.55), respectively.

We now claim that e_p is noncritical in $H' \circ \langle e_p \rangle$. Assume, to the contrary, that e_p is critical in $H' \circ \langle e_p \rangle$. Apply Lemma 5.2, with 'H' $\leftarrow H'$, 'H'' $\leftarrow H$ (*i.e.*, with H and H' interchanged), 'G' $\leftarrow \langle\rangle$, and '$e'_p$' $\leftarrow e_p$. Among the assumptions stated in Lemma 5.2, (B.4)–(B.7), (B.9), (B.10) follow from (B.57), (B.49), (B.52), (B.51), (B.56), and (B.54), respectively; (B.8) is trivial; (B.11) and (B.12) follow from $G = \langle\rangle$;

(B.13) follows from our assumption that e_p is critical in $H' \circ \langle e_p \rangle$. From the lemma, it follows that e_p is a critical event in $H \circ \langle e_p \rangle$, a contradiction. Therefore,

- e_p is noncritical in $H' \circ \langle e_p \rangle$. (B.58)

Finally, by applying Lemma B.1 with 'H' $\leftarrow H'$, it follows that RFS is a valid RF-set of $H' \circ \langle e_p \rangle$. Assumptions (B.36)–(B.39) stated in Lemma B.1 follow from (B.57), (B.52), (B.56), and (B.58), respectively. □

Corollary B.2 Consider two computations H and H', two sets of processes RFS and Z, and another computation L. Assume the following:

- $H \circ L \in C$; (B.59)
- $H' \in C$; (B.60)
- RFS is a valid RF-set of H; (B.61)
- RFS is a valid RF-set of H'; (B.62)
- $Z \subseteq \text{Act}(H)$; (B.63)
- $H \mid (Z \cup RFS) = H' \mid (Z \cup RFS)$; (B.64)
- L is a Z-computation; (B.65)
- L has no critical events in $H \circ L$. (B.66)

Then, the following hold: $H' \circ L \in C$; RFS is a valid RF-set of both $H \circ L$ and $H' \circ L$; L has no critical events in $H' \circ L$.

Proof: The proof of Corollary B.2 easily follows by induction on the length of L, applying Lemma B.2 at each induction step. □

Lemma 5.4 Consider a computation H, a set of processes RFS, and another set of processes $Y = \{p_1,\ p_2,\ \ldots,\ p_m\}$. Assume the following:

- $H \in C$; (B.67)
- RFS is a valid RF-set of H; (B.68)
- $Y \subseteq \text{Inv}_{RFS}(H)$; (B.69)
- for each p_j in Y, there exists a computation L_{p_j}, satisfying the following:
  - L_{p_j} is a p_j-computation; (B.70)
  - $H \circ L_{p_j} \in C$; (B.71)
  - L_{p_j} has no critical events in $H \circ L_{p_j}$. (B.72)

Define L to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_m}$. Then, the following hold: $H \circ L \in C$, RFS is a valid RF-set of $H \circ L$, and L contains no critical events in $H \circ L$.

Proof: First, note that (B.69) implies

$$Y \subseteq \mathrm{Act}(H). \tag{B.73}$$

Define $L^0 = \langle\rangle$; for each positive j, define L^j to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_j}$. We prove the lemma by induction on j. At each step, we assume

- $H \circ L^j \in C$, (B.74)
- RFS is a valid RF-set of $H \circ L^j$, and (B.75)
- L^j contains no critical events in $H \circ L^j$. (B.76)

The induction base ($j = 0$) follows easily from (B.67) and (B.68), since $L^0 = \langle\rangle$.

Assume that (B.74)–(B.76) hold for a particular value of j. By definition, $L^j \mid p_{j+1} = \langle\rangle$, and hence

$$H \mid (\{p_{j+1}\} \cup \mathit{RFS}) = (H \circ L^j) \mid (\{p_{j+1}\} \cup \mathit{RFS}). \tag{B.77}$$

We use Corollary B.2, with 'H'' $\leftarrow H \circ L^j$, 'Z' $\leftarrow \{p_{j+1}\}$, and 'L' $\leftarrow L_{p_{j+1}}$. Among the assumptions stated in Corollary B.2, (B.60)–(B.62) and (B.64) follow from (B.74), (B.68), (B.75), and (B.77), respectively; (B.63) follows from (B.73) and $p_{j+1} \in Y$; (B.59), (B.65), and (B.66) follow from (B.71), (B.70), and (B.72), respectively, each applied with 'p_j' $\leftarrow p_{j+1}$. This gives us the following:

- $H \circ L^{j+1} = (H \circ L^j) \circ L_{p_{j+1}} \in C$;
- RFS is a valid RF-set of $H \circ L^{j+1}$;
- $L_{p_{j+1}}$ contains no critical events in $H \circ L^{j+1}$. (B.78)

By (B.76) and (B.78), it follows that L^{j+1} contains no critical events in $H \circ L^{j+1}$. □

Lemma 5.5 Let H be a computation. Assume the following:

- $H \in C$, and (B.79)
- H is regular (i.e., $\mathrm{Fin}(H)$ is a valid RF-set of H). (B.80)

Define $n = |\mathrm{Act}(H)|$. Then, there exists a subset Y of $\mathrm{Act}(H)$, where $n - 1 \le |Y| \le n$, satisfying the following: for each process p in Y, there exist a p-computation L_p and an event e_p by p such that

- $H \circ L_p \circ \langle e_p\rangle \in C$; (B.81)
- L_p contains no critical events in $H \circ L_p$; (B.82)
- $e_p \notin \{\mathit{Enter}_p, \mathit{CS}_p, \mathit{Exit}_p\}$; (B.83)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (B.84)

- e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$. (B.85)

Proof: First, we construct, for each process p in $\mathrm{Act}(H)$, a computation L_p and an event e_p that satisfy (B.81) and (B.82). Then, we show that every event e_p thus constructed, except at most one, satisfies (B.83). The other conditions can be easily proved thereafter.

For each process p in $\mathrm{Act}(H)$, define H_p as

$$H_p = H \mid (\{p\} \cup \mathrm{Fin}(H)). \tag{B.86}$$

We apply Lemma 5.1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow \{p\} \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 5.1, (B.1) and (B.2) follow from (B.79) and (B.80), respectively; (B.3) is trivial. It follows that H_p is in C and

- $\mathrm{Fin}(H)$ is a valid RF-set of H_p. (B.87)

Since $p \in \mathrm{Act}(H)$, by (B.86), we have

$$\mathrm{Act}(H_p) = \{p\} \quad \wedge \quad \mathrm{Fin}(H_p) = \mathrm{Fin}(H). \tag{B.88}$$

Thus, by (B.87), and applying RF4 to H, we have

$$value(stat_q, H_p) = \begin{cases} ncs, & \text{for all } q \neq p \\ entry, & \text{for } q = p. \end{cases}$$

Therefore, by the Progress property, there exists a p-computation F_p such that $H_p \circ F_p \circ \langle CS_p \rangle \in C$. If F_p has a critical event in $H_p \circ F_p \circ \langle CS_p \rangle$, then let e'_p be the first critical event in F_p, and let L_p be the prefix of F_p that precedes e'_p (*i.e.*, $F_p = L_p \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L_p to be F_p and e'_p to be CS_p. By P1, we have $H_p \circ L_p \circ \langle e'_p \rangle \in C$ and $H_p \circ L_p \in C$.

We have just constructed a computation L_p and an event e'_p by p, such that

- $H_p \circ L_p \circ \langle e'_p \rangle \in C$, (B.89)
- $H_p \circ L_p \in C$, (B.90)
- L_p is a p-computation, (B.91)
- L_p has no critical events in $H_p \circ L_p$, and (B.92)
- e'_p is a critical event in $H_p \circ L_p \circ \langle e'_p \rangle$. (B.93)

The following assertion follows easily from (B.86).

$$(H_p \circ L_p) \mid (\{p\} \cup \mathrm{Fin}(H)) = (H \circ L_p) \mid (\{p\} \cup \mathrm{Fin}(H)). \qquad \text{(B.94)}$$

We now use Corollary B.2, with 'H' $\leftarrow H_p$, 'H'' $\leftarrow H$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Z$' $\leftarrow \{p\}$, and 'L' $\leftarrow L_p$. Assumptions (B.59)–(B.66) stated in Corollary B.2 follow from (B.90), (B.79), (B.87), (B.80), (B.88), (B.94), (B.91), and (B.92), respectively. Thus, we have the following:

- $H \circ L_p \in C$; (B.95)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H_p \circ L_p$; (B.96)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (B.97)
- L_p has no critical events in $H \circ L_p$, which establishes (B.82).

Because $H \circ L_p$ and $H_p \circ L_p$ are equivalent with respect to p, by (B.89), (B.95), and P3, there exists an event e_p of p such that

- $e_p \sim e'_p$, and (B.98)
- $H \circ L_p \circ \langle e_p \rangle \in C$, which establishes (B.81).

We now claim that at most one process in $\mathrm{Act}(H)$ fails to satisfy (B.83). Because $p \in \mathrm{Act}(H)$ and H is regular, by RF4, $value(stat_p, H) = entry$ holds. Thus, by the definition of a mutual exclusion system, e_p cannot be $Enter_p$ or $Exit_p$. It suffices to show that there can be at most one process in $\mathrm{Act}(H)$ such that $e_p = CS_p$.

Assume, to the contrary, that there are two distinct processes p and q in $\mathrm{Act}(H)$, such that $e_p = CS_p$ and $e_q = CS_q$. Note that (B.80) implies that $\mathrm{Inv}_{\mathrm{Fin}(H)}(H) = \mathrm{Act}(H)$, and hence

$$\{p, q\} \subseteq \mathrm{Inv}_{\mathrm{Fin}(H)}(H). \qquad \text{(B.99)}$$

By applying Lemma 5.4 with 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow \{p, q\}$, 'L_{p_1}' $\leftarrow L_p$, and 'L_{p_2}' $\leftarrow L_q$, we have $H \circ L_p \circ L_q \in C$. Among the assumptions stated in Lemma 5.4, (B.67)–(B.69) follow from (B.79), (B.80), and (B.99), respectively; (B.70)–(B.72) follow from (B.91), (B.95), (B.82), respectively, each with 'p' $\leftarrow p$ and then 'p' $\leftarrow q$.

Since $H \circ L_p$ is equivalent to $H \circ L_p \circ L_q$ with respect to p, and since CS_p does not read any variable, by P2, we have $H \circ L_p \circ L_q \circ \langle CS_p \rangle \in C$. Similarly, we also have $H \circ L_p \circ L_q \circ \langle CS_q \rangle \in C$. Hence $H \circ L_p \circ L_q$ violates the Exclusion property, a contradiction.

Therefore, there exists a subset Y of $\mathrm{Act}(H)$ such that $n-1 \leq |Y| \leq n$ and each process in Y satisfies (B.81), (B.82), and (B.83).

We claim that each process p in Y satisfies (B.84). Note that, since $p \in Y$ and $Y \subseteq \mathrm{Act}(H)$, by (B.91),

- L_p is a Y-computation. (B.100)

Condition (B.84) now follows from Corollary B.1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$L$' $\leftarrow L_p$. Assumptions (B.45)–(B.48) stated in the corollary follow from (B.95), (B.80), (B.100), and (B.82), respectively.

Finally, we prove (B.85). Note that, by (B.88), (B.91), and (B.92), we have

$$\mathrm{Act}(H_p \circ L_p) = \{p\}. \tag{B.101}$$

Condition (B.85) now follows from Lemma 5.2, with 'H' $\leftarrow H_p \circ L_p$, 'H'' $\leftarrow H \circ L_p$, 'G' $\leftarrow \langle\rangle$, '$RFS$' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e'_p$, and 'e'_p' $\leftarrow e_p$. Assumptions (B.4)–(B.13) stated in Lemma 5.2 follow from (B.89), (B.81), (B.96), (B.97), (B.98), (B.101), (B.94), $G = \langle\rangle$, $G = \langle\rangle$, and (B.93), respectively. □

Lemma 5.6 Consider a computation H and set of processes RFS. Assume the following:

- $H \in C$; (B.102)
- RFS is a valid RF-set of H; (B.103)
- $\mathrm{Fin}(H) \subsetneq RFS$ (*i.e.*, $\mathrm{Fin}(H)$ is a proper subset of RFS). (B.104)

Then, there exists a computation G satisfying the following.

- $G \in C$; (B.105)
- RFS is a valid RF-set of G; (B.106)
- G can be written as $H \mid (Y \cup RFS) \circ L \circ \langle e_p \rangle$, for some choice of Y, L, and e_p, satisfying the following:
  - Y is a subset of $\mathrm{Inv}(H)$ such that $|\mathrm{Inv}(H)| - 1 \leq |Y| \leq |\mathrm{Inv}(H)|$, (B.107)
  - $\mathrm{Inv}(G) = Y$, (B.108)
  - L is a $\mathrm{Pmt}(H)$-computation, (B.109)
  - L has no critical events in G, (B.110)
  - $p \in \mathrm{Pmt}(H)$, and (B.111)
  - e_p is critical in G; (B.112)
- $\mathrm{Pmt}(G) \subseteq \mathrm{Pmt}(H)$; (B.113)

- An event in $H \mid (Y \cup \mathit{RFS})$ is critical if and only if it is also critical in H. (B.114)

Proof: Define $Z = \mathrm{Pmt}(H)$. Then, by definition, $Z \subseteq \mathrm{Act}(H)$. Define H' as

$$H' = H \mid \mathit{RFS}. \tag{B.115}$$

Apply Lemma 5.1, with 'Y' $\leftarrow \mathit{RFS}$. Assumptions (B.1)–(B.3) stated in Lemma 5.1 follow from (B.102)–(B.104), respectively.

- RFS is a valid RF-set of H'. (B.116)

Also, by (B.115), we have

$$\mathrm{Act}(H') = Z \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{B.117}$$

Therefore, by (B.116), and applying RF4 to H', we have

$$\mathit{value}(\mathit{stat}_q, H') = \begin{cases} \mathit{entry} \text{ or } \mathit{exit}, & \text{if } q \in Z, \\ \mathit{ncs}, & \text{if } q \notin Z. \end{cases}$$

Therefore, by the Progress property, there exists a Z-computation F such that $H' \circ F \circ \langle f_r \rangle \in C$, where r is a process in Z and f_r is either CS_r or Exit_r. If F has a critical event in $H' \circ F \circ \langle f_r \rangle$, , then let e'_p be the first critical event in F, and let L be the prefix of F that precedes e'_p (*i.e.*, $F = L \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L to be F and e'_p to be f_r. By P1, we have $H' \circ L \circ \langle e'_p \rangle \in C$ and $H' \circ L \in C$. Because F is a Z-computation, we have $p \in Z$, which implies

- $p \in Z \subseteq \mathrm{Act}(H)$, and (B.118)
- $p \in \mathit{RFS}$. (B.119)

We have just constructed a computation L and an event e'_p by p, such that

- $H' \circ L \circ \langle e'_p \rangle \in C$, (B.120)
- $H' \circ L \in C$, (B.121)
- L is a Z-computation, (B.122)
- L has no critical events in $H' \circ L$, and (B.123)
- e'_p is a critical event in $H' \circ L \circ \langle e'_p \rangle$. (B.124)

The following assertion follows easily from (B.115) and (B.122). (Note that $Z \subseteq RFS$ holds by definition.)

$$(H' \circ L) \mid RFS = (H \circ L) \mid RFS. \tag{B.125}$$

We now use Corollary B.2, with 'H' $\leftarrow H'$ and 'H'' $\leftarrow H$. Among the assumptions stated in Corollary B.2, (B.59)–(B.63), (B.65), and (B.66) follow from (B.121), (B.102), (B.116), (B.103), (B.117), (B.122), and (B.123), respectively; (B.64) follows from (B.125) and $Z \subseteq RFS$. Thus, we have the following:

- $H \circ L \in C$; (B.126)
- RFS is a valid RF-set of $H' \circ L$; (B.127)
- RFS is a valid RF-set of $H \circ L$; (B.128)
- L has no critical events in $H \circ L$. (B.129)

Note that, by (B.122), (B.129), and $Z \subseteq \mathrm{Act}(H)$, we have

$$\mathrm{Act}(H \circ L) = \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H \circ L) = \mathrm{Fin}(H). \tag{B.130}$$

In particular, by (B.118),

$$p \in \mathrm{Act}(H \circ L). \tag{B.131}$$

Because $H \circ L$ and $H' \circ L$ are equivalent with respect to p, by (B.120), (B.126), and P3, there exists an event e''_p of p such that

- $e''_p \sim e'_p$, and (B.132)
- $H \circ L \circ \langle e''_p \rangle \in C$. (B.133)

We now use Lemma 5.2, with 'H' $\leftarrow H' \circ L$, 'H'' $\leftarrow H \circ L$, 'G' $\leftarrow \langle\rangle$, '$e_p$' $\leftarrow e'_p$, and 'e'_p' $\leftarrow e''_p$. Among the assumptions stated in Lemma 5.2, (B.4)–(B.8) and (B.11)–(B.13) follow from (B.120), (B.133), (B.127), (B.128), (B.132), 'G' $\leftarrow \langle\rangle$, '$G$' $\leftarrow \langle\rangle$, and (B.124), respectively; (B.9) follows from (B.118) and (B.123); (B.10) follows from (B.119) and (B.125). It follows that

- e''_p is critical in $H \circ L \circ \langle e''_p \rangle$. (B.134)

We now establish (B.105)–(B.114) by considering two cases separately.

Case 1: e''_p is a transition event. In this case, define $Y = \mathrm{Inv}(H)$, $e_p = e''_p$, and $G = H \circ L \circ \langle e_p \rangle$. Note that, by (B.103), we have

$$H = H \mid (Y \cup RFS) \quad \wedge \quad G = H \mid (Y \cup RFS) \circ L \circ \langle e_p \rangle. \tag{B.135}$$

We claim that these definitions satisfy (B.105)–(B.114). Conditions (B.105) and (B.109)–(B.111) follow from (B.133), (B.122), (B.129), and (B.118), respectively. Condition (B.107) is trivial.

We now establish (B.106). Before proving RF1–RF5, we need to prove that $\mathrm{Fin}(G) \subseteq RFS$ and that $G \mid q \neq \langle\rangle$ holds for each $q \in RFS$. Condition (B.128) implies $\mathrm{Fin}(H \circ L) \subseteq RFS$. By the definition of G, we have $\mathrm{Fin}(G) \subseteq \mathrm{Fin}(H \circ L) \cup \{p\}$. Thus, by (B.119), we have $\mathrm{Fin}(G) \subseteq RFS$. Condition (B.128) also implies that $G \mid q \neq \langle\rangle$ holds for each $q \in RFS$. We now check each of RF1–RF5.

- **RF1, RF2, and RF3:** Since appending a transition event does not invalidate any of RF1, RF2, and RF3, it easily follows, by (B.128), that G satisfies these conditions.

- **RF4:** By (B.128), it suffices to show that p satisfies RF4, which follows easily from (B.119) and (B.131). In particular, if $e_p = CS_p$, then $p \in \mathrm{Pmt}(G)$ and $value(stat_p, G) = exit$ hold; if $e_p = Exit_p$, then $p \in \mathrm{Fin}(G)$ and $value(stat_p, G) = ncs$ hold. (Note that e_p cannot be $Enter_p$, because by (B.128) and (B.131), and applying RF4 to p in $H \circ L$, we have $value(stat_p, H \circ L) \neq ncs$.)

- **RF5:** It suffices to show that e_p is also a critical event in $G \mid RFS$. However, since e_p is a transition event, this is immediate.

It follows that (B.106) holds.

We now conclude Case 1 by establishing (B.108) and (B.112)–(B.114). By (B.130), we have $\mathrm{Inv}(H \circ L) = (\mathrm{Act}(H \circ L) - RFS) = (\mathrm{Act}(H) - RFS) = \mathrm{Inv}(H)$. Moreover, (B.119) implies that appending e_p to $H \circ L$ cannot change the set of invisible processes. Thus, we have (B.108). Condition (B.112) holds by definition, since e_p is a transition event. In order to prove (B.113), note that any process in $\mathrm{Fin}(H)$ is also in $\mathrm{Fin}(G)$ by the definition of a finished process. Thus, we have $\mathrm{Pmt}(G) = (RFS - \mathrm{Fin}(G)) \subseteq (RFS - \mathrm{Fin}(H)) = Z$. Finally, (B.135) implies that (B.114) is trivially true.

Case 2: e''_p is *not* a transition event. In this case, there exists a variable v_{ce} (for "critical event"), remote to p, that is accessed by e''_p. If v_{ce} is local to a process in $\mathrm{Inv}(H)$, let x_{loc} be the process that v_{ce} is local to; otherwise, let $x_{\mathrm{loc}} = \perp$. Similarly, if $writer(v_{\mathrm{ce}}, H \circ L) \in \mathrm{Inv}(H)$ holds, let $x_w = writer(v_{\mathrm{ce}}, H \circ L)$ and $f_{x_w} = writer\_event(v_{\mathrm{ce}}, H \circ L)$; otherwise, let $x_w = \perp$ and $f_{x_w} = \perp$. By definition,

- if $x_{\text{loc}} \neq \perp$, then $x_{\text{loc}} \in \text{Inv}(H) \subseteq \text{Act}(H)$. (B.136)
- if $x_w \neq \perp$, then $x_w \in \text{Inv}(H) \subseteq \text{Act}(H)$. (B.137)

We now establish the following simple claim.

Claim 1: If $x_{\text{loc}} \neq \perp$ and $x_w \neq \perp$, then $x_{\text{loc}} = x_w$.

Proof of Claim: Assume, to the contrary, that $x_{\text{loc}} \neq \perp$, $x_w \neq \perp$, and $x_{\text{loc}} \neq x_w$ hold. Then, f_{x_w} *remotely* writes to v_{ce}. Hence, by (B.128), and applying RF2 to f_{x_w} in $H \circ L$, we have either $x_{\text{loc}} \notin \text{Act}(H \circ L)$ or $\{x_w, x_{\text{loc}}\} \in \mathit{RFS}$. However, by (B.130) and (B.136), we have $x_{\text{loc}} \in \text{Act}(H \circ L)$ and $x_{\text{loc}} \notin \mathit{RFS}$, a contradiction. (Note that, by (B.103), $x_{\text{loc}} \in \text{Inv}(H)$ implies $x_{\text{loc}} \notin \mathit{RFS}$.) $\square$

We now define Y as follows. By Claim 1, Y is well-defined.

$$Y = \begin{cases} \text{Inv}(H) - \{x_{\text{loc}}\}, & \text{if } x_{\text{loc}} \neq \perp; \\ \text{Inv}(H) - \{x_w\}, & \text{if } x_w \neq \perp; \\ \text{Inv}(H), & \text{if } x_{\text{loc}} = \perp \text{ and } x_w = \perp. \end{cases} \tag{B.138}$$

Let $\overline{G} = H \mid (Y \cup \mathit{RFS}) \circ L$. (Informally, $\overline{G}$ is a computation that is obtained by erasing x_{loc} and x_w from H. By erasing x_{loc}, we preserve RF2. By erasing x_w, we preserve RF3 and eliminate potential information flow. Since L has no critical events in $\overline{G}$ [see (B.149) below], appending L does not create information flow.) We now establish a number of assertions concerning $\overline{G}$, after which we define G. By (B.122) and $Z \subseteq \mathit{RFS}$, we have

$$\overline{G} = H \mid (Y \cup \mathit{RFS}) \circ L = (H \circ L) \mid (Y \cup \mathit{RFS}). \tag{B.139}$$

We now apply Lemma 5.1, with 'H' $\leftarrow H \circ L$ and 'Y' $\leftarrow Y \cup \mathit{RFS}$. Among the assumptions stated in Lemma 5.1, (B.1) and (B.2) follow from (B.126) and (B.128), respectively; (B.3) is trivial. Thus, we have the following:

- $\overline{G}$ is in C, (B.140)
- RFS is a valid RF-set of $\overline{G}$, and (B.141)
- an event in $\overline{G}$ is critical if and only if it is also critical in $H \circ L$. (B.142)

By (B.119), we have $\mathit{RFS} = \{p\} \cup \mathit{RFS}$. Thus, by (B.131) and (B.139), we have the following:

- $p \in \mathrm{Act}(\overline{G})$, and (B.143)
- $\overline{G} \mid (\{p\} \cup RFS) = (H \circ L) \mid (\{p\} \cup RFS)$. (B.144)

If $x_{\mathrm{loc}} \neq \bot$, then we have $x_{\mathrm{loc}} \notin Y$ by (B.138), and also $x_{\mathrm{loc}} \notin RFS$, since $x_{\mathrm{loc}} \in \mathrm{Inv}(H)$. Thus, by (B.139), it follows that

- $\overline{G} \mid x_{\mathrm{loc}} = \langle\rangle$, if $x_{\mathrm{loc}} \neq \bot$. (B.145)

Similarly,

- $\overline{G} \mid x_w = \langle\rangle$, if $x_w \neq \bot$. (B.146)

By (B.144), $\overline{G}$ is equivalent to $H \circ L$ with respect to p. Therefore, by (B.133), (B.140), and P3, there exists an event e_p such that

- $e_p \sim e''_p$, and (B.147)
- $\overline{G} \circ \langle e_p \rangle \in C$. (B.148)

Define G to be $\overline{G} \circ \langle e_p \rangle = H \mid (Y \cup RFS) \circ L \circ \langle e_p \rangle$. We claim that G satisfies the lemma. To show this, we need a few additional assertions. By (B.129), (B.139), and (B.142), it follows that

- L has no critical events in $\overline{G}$, and (B.149)
- an event in $H \mid (Y \cup RFS)$ is critical if and only if it is also critical in H. (B.150)

Also, by (B.143), and since e_p is not a transition event,

$$\mathrm{Act}(G) = \mathrm{Act}(\overline{G}). \tag{B.151}$$

We now prove that G satisfies the lemma. Each of the conditions (B.105), (B.107), (B.109), (B.111), and (B.114) follows easily from (B.148), (B.138), (B.122), (B.118), and (B.150), respectively. Since $\overline{G}$ is a prefix of G, (B.110) follows from (B.149). By (B.138), (B.139), and (B.149), the active processes in $\overline{G}$ include those in Y, which are invisible, and any promoted processes in RFS; hence, $\mathrm{Act}(\overline{G}) - RFS = Y - RFS = Y$. Moreover, the processes in RFS that are active in $H \circ L$ are also active in $\overline{G}$; hence, $\mathrm{Act}(\overline{G}) \cap RFS = \mathrm{Act}(H \circ L) \cap RFS$. Thus, by (B.130) and (B.151), we have

$$\begin{aligned}\mathrm{Inv}(G) &= \mathrm{Act}(G) - RFS = \mathrm{Act}(\overline{G}) - RFS \\ &= Y - RFS = Y,\end{aligned}$$

and

$$\begin{aligned}\mathrm{Pmt}(G) &= \mathrm{Act}(G) \cap \mathit{RFS} = \mathrm{Act}(\overline{G}) \cap \mathit{RFS} \\ &= \mathrm{Act}(H \circ L) \cap \mathit{RFS} = \mathrm{Act}(H) \cap \mathit{RFS} = Z,\end{aligned}$$

which imply (B.108) and (B.113).

In order to prove (B.112), we apply Lemma 5.2 with 'H' $\leftarrow H \circ L$, 'H'' $\leftarrow \overline{G}$, 'G' $\leftarrow \langle\rangle$, '$e_p$' $\leftarrow e''_p$, and 'e'_p' $\leftarrow e_p$. Assumptions (B.4)–(B.13) stated in Lemma 5.2 follow from (B.133), (B.148), (B.128), (B.141), (B.147), (B.131), (B.144), 'G' $\leftarrow \langle\rangle$, '$G$' $\leftarrow \langle\rangle$, and (B.134), respectively. Moreover, Assumption (A) follows from (B.141), and (B) holds vacuously by 'G' $\leftarrow \langle\rangle$. It follows that e_p is critical in G, *i.e.*, (B.112) holds, and

- G satisfies RF5. (B.152)

This leaves only (B.106) to be proved. Let $x' = \mathit{writer}(v_{\mathrm{ce}}, \overline{G})$ and $f_{x'} = \mathit{writer\_event}(v_{\mathrm{ce}}, \overline{G})$. We begin by establishing the following claim.

Claim 2: $x' = \perp$ or $x' \in \mathit{RFS}$.

Proof of Claim: Assume, to the contrary, that $x' \neq \perp$ and $x' \notin \mathit{RFS}$ hold. If $f_{x'}$ is an event of L, then by (B.122) and the definition of Z, we have $x' \in Z \subseteq \mathit{RFS}$, a contradiction. Thus, $f_{x'}$ is an event of $H \mid (Y \cup \mathit{RFS})$. Since we assumed $x' \notin \mathit{RFS}$, by (B.138), we have $x' \in Y \subseteq \mathrm{Inv}(H)$. If $x' = \mathit{writer}(v_{\mathrm{ce}}, H \circ L)$, then since $x' \in \mathrm{Inv}(H)$, we have $x' = x_w$ by the definition of x_w, which is impossible by (B.146). Thus, we have $x' \neq \mathit{writer}(v_{\mathrm{ce}}, H \circ L)$, which implies

$$x_w \neq x'. \tag{B.153}$$

Note that, by (B.137) and (B.130), we have

$$x_w \neq \perp \;\Rightarrow\; x_w \in \mathrm{Act}(H \circ L). \tag{B.154}$$

Also, since $x' \in \mathrm{Inv}(H) \subseteq \mathrm{Act}(H)$, by (B.130),

$$x' \in \mathrm{Act}(H \circ L). \tag{B.155}$$

We now show that $\mathit{writer}(v_{\text{ce}}, H \circ L) \in \mathit{RFS}$ holds. If $x_w = \bot$, then since there exists a write to v (*i.e.*, $f_{x'}$) in $H \circ L$, the definition of x_w implies $\mathit{writer}(v_{\text{ce}}, H \circ L) \notin \text{Inv}(H)$, which in turn implies $\mathit{writer}(v_{\text{ce}}, H \circ L) \in \mathit{RFS}$ by (B.128) and (B.130). On the other hand, if $x_w \neq \bot$, then by (B.128), (B.153), (B.154), (B.155), and by applying RF3 to $f_{x'}$ and f_{x_w} in $H \circ L$, we have $\mathit{writer}(v_{\text{ce}}, H \circ L) \in \mathit{RFS}$.

Because $\mathit{writer}(v_{\text{ce}}, H \circ L) \in \mathit{RFS}$ holds, by (B.139), the last event to write to v_{ce} is identical in $H \circ L$ and $\overline{G}$. Thus, we have $x' \in \mathit{RFS}$, a contradiction. □

We now establish (B.106) by showing that RFS is a valid RF-set of G. Condition RF5 was already proved in (B.152). Before proving RF1–RF4, we need to prove that $\text{Fin}(G) \subseteq \mathit{RFS}$ and that $G \mid q \neq \langle\rangle$ holds for each $q \in \mathit{RFS}$. Condition (B.141) implies $\text{Fin}(\overline{G}) \subseteq \mathit{RFS}$. Since e_p is not a transition event, we have $\text{Fin}(G) = \text{Fin}(\overline{G})$, and hence $\text{Fin}(G) \subseteq \mathit{RFS}$. Condition (B.141) also implies that $G \mid q \neq \langle\rangle$ holds for each $q \in \mathit{RFS}$. We now check each of RF1–RF4.

- **RF1 and RF2:** We use Lemma 5.3 to prove these two conditions. First, we need the following claim.

  Claim 3: If v_{ce} is local to a process q, then either $q \notin \text{Act}(\overline{G})$ or $\{p, q\} \subseteq \mathit{RFS}$ holds.

  Proof of Claim: If v_{ce} is local to a process q, then one of the following holds: $q \in \text{Inv}(H)$, $q \in \mathit{RFS}$, or $H \mid q = \langle\rangle$. If $q \in \text{Inv}(H)$, then $q = x_{\text{loc}}$ by definition, and hence, by (B.145), we have $q \notin \text{Act}(\overline{G})$. If $q \in \mathit{RFS}$, then by (B.119), we have $\{p, q\} \subseteq \mathit{RFS}$. By (B.130) and (B.139), we have $\text{Act}(\overline{G}) \subseteq \text{Act}(H \circ L) = \text{Act}(H)$, and hence $H \mid q = \langle\rangle$ implies $q \notin \text{Act}(\overline{G})$. □

  We now use Lemma 5.3, with 'H' $\leftarrow \overline{G}$, 'G' $\leftarrow \langle\rangle$, and '$v_{\text{rem}}$' $\leftarrow v_{\text{ce}}$. Among the assumptions stated in Lemma 5.3, assumptions (B.25)–(B.28) follow from (B.148), (B.141), (B.143), and (B.141), respectively; (B.29) and (B.30) are trivial; (B.31) follows from Claim 3; (B.32) (with 'q' $\leftarrow x'$) follows from Claim 2. It follows that G satisfies RF1 and RF2.

- **RF3:** Consider a variable $v \in V$ and two different events g_q and h_r in G. Assume that both q and r are in $\mathrm{Act}(G)$, $q \neq r$, that there exists a variable v such that $v \in \mathit{var}(g_q) \cap \mathit{var}(h_r)$, and that there exists a write to v in G. Define $s = \mathit{writer}(v, G)$. Our proof obligation is to show that $s \in \mathit{RFS}$.

  Since $p \in Z \subseteq \mathit{RFS}$, it suffices to consider the case in which $s \neq p$, in which case we also have the following: e_p does not write v, $s = \mathit{writer}(v, \overline{G})$, and there exists a write to v in $\overline{G}$.

  - First, consider the case in which both g_q and h_r are in $\overline{G}$. By (B.151), we have $q \in \mathrm{Act}(\overline{G})$ and $r \in \mathrm{Act}(\overline{G})$. Thus, by (B.141), and by applying RF3 to g_q and h_r in $\overline{G}$, we have $s \in \mathit{RFS}$.
  - Second, consider the case in which one of g_q or h_r is e_p. Without loss of generality, we can assume that g_q is in $\overline{G}$ and $h_r = e_p$. Then, we have $q \neq p$ and $r = p$. If v is local to p, then by (B.141), (B.143), $s \neq p$, and by applying RF2 to $\mathit{writer\_event}(v, \overline{G})$ by s in $\overline{G}$, we have $s \in \mathit{RFS}$. On the other hand, if v is remote to p, then by the Atomicity property, we have $v = v_{\mathrm{ce}}$ and $s = \mathit{writer}(v_{\mathrm{ce}}, \overline{G}) = x'$, in which case, by Claim 2, we have $s = \bot$ or $s \in \mathit{RFS}$. Since there exists a write to v_{ce} in $\overline{G}$ by assumption, we have $s \neq \bot$, and hence $s \in \mathit{RFS}$.

- **RF4:** Since $\overline{G}$ satisfies RF4 by (B.141), and since e_p is not a transition event, RF4 follows trivially. $\square$

APPENDIX C

CORRECTNESS PROOF FOR ALGORITHM NA IN Chapter 7

In this appendix, we present a detailed correctness proof of ALGORITHM NA, presented in Chapter 7. To simplify the proof, we consider a recursive version of ALGORITHM NA, shown in Figure C.1. In particular, only the root node (statements 2–20) is presented in detail, and the details of the left and right subtrees beneath the root are hidden in statements 1 and 21. As shown in [12], a nonatomic algorithm can be converted into an equivalent atomic algorithm by assuming that all reads execute atomically, and by replacing each nonatomic write $v := val$ (where val is an expression over private variables) by the following code fragment.[1]

```
L:  flag := (a nondeterministically selected boolean value);
    if flag then
        v := (a nondeterministically selected value over the domain of v);
        goto L
    else
        v := val
    fi
```

We assume that the semantics of ALGORITHM NA is defined in this way. For example, in Figure C.1, if process p executes statement 5 while $p.rtoggle = true$ holds, then it may establish one of the following three conditions: **(i)** $R2[p] = true \wedge p@\{5\}$, **(ii)** $R2[p] = false \wedge p@\{5\}$, or **(iii)** $R2[p] = true \wedge p@\{6\}$. Statements 7, 11, and 20 also have similar properties. Note that these four statements are the only writes of nonatomic variables in ALGORITHM NA, since variables T and C are implemented

[1]If a variable is written by multiple processes simultaneously, then this code fragment may not represent the semantics of the nonatomic variable, depending on the system model. However, in our nonatomic algorithm, all multi-writer variables are implemented via wait-free register constructions, in which an atomic variable is implemented using only nonatomic variables. Thus, we may assume that all nonatomic variables are single-writer variables.

```
shared variables
    $T$: $0..N-1$;
    $C$: array[0..1] of $(0..N-1,\ \perp)$ initially $\perp$;
    $Q1, Q2, R1, R2$: array[$0..N-1$] of boolean

private variables
    $qtoggle, rtoggle, temp$: boolean;
    $rival$: $0..N-1$

private constant
    $side$ = if $p < N/2$ then 0 else 1 fi

process $p$ ::    /* $0 \le p < N$ */
while true do
0:  Noncritical Section;
1:  if $side = 0$ then
        (enter the left subtree)
    else
        (enter the right subtree)
    fi
2:  $C[side] := p$;
3:  $T := p$;
4:  $rtoggle := \neg R1[p]$;
5:  $R2[p] := rtoggle$;
6:  $qtoggle := \neg Q1[p]$;
7:  $Q2[p] := qtoggle$;
8:  $rival := C[1 - side]$;
    if $(rival \neq \perp\ \wedge$
9:          $T = p)$ then
10:     $temp := Q2[rival]$;
11:     $Q1[rival] := temp$;
12:     await $(Q1[p] = qtoggle) \vee$
13:                 $(R1[p] = rtoggle)$;
14:     if $T = p$ then
15:         await $R1[p] = rtoggle$ fi
    fi;
16: Critical Section;
17: $C[side] := \perp$;
18: $rival := T$;
    if $rival \neq p$ then
19:     $temp := R2[rival]$;
20:     $R1[rival] := temp$
    fi;
21: if $side = 0$ then
        (exit the left subtree)
    else
        (exit the right subtree)
    fi
od
```

Figure C.1: Recursive version of ALGORITHM NA. Only statements 5, 7, 11, and 20 are executed nonatomically.

using register constructions. Other than statements 5, 7, 11, and 20, we assume that each labeled sequence of statements in Figure C.1 is atomic. Note that each numbered sequence of statements (except statements 1 and 21, which are considered below) reads or writes at most one shared variable.

We establish the correctness of ALGORITHM NA by induction on the level of the tree. That is, we prove that if ALGORITHM NA is correct for an arbitration tree with l levels, then it is also correct for an arbitration tree with $l+1$ levels. By induction, we may assume that the left and the right subtrees (statements 1 and 21) are correct mutual exclusion algorithms, and hence we may assume that statements 1 and 21 execute atomically. Moreover, we have the following invariant.

invariant $\left|\{p :: p@\{2..21\} \ \wedge \ p.side = s\}\right| \leq 1$ (I0)

Invariant (I0) states that at most one process may execute statements 2–21 from either side at any time. Thus, we need to consider at most two processes executing in these statements at any given state.

We now prove that each of invariants (I1)–(I16), stated below, is an invariant. Invariant (I6) establishes the Exclusion property; invariants (I7)–(I16) are used to prove starvation-freedom. (Many of these invariants are adapted from [84].) For each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.) If I is an implication (which is the case for most of our invariants), then it suffices to check only those program statements that may establish the antecedent of I, or that may falsify the consequent if executed while the antecedent holds.

invariant $\big(C[s] = p \ \wedge \ p \neq \bot\big) = \big(p@\{3..17\} \ \wedge \ p.side = s\big)$ (I1)

Proof: The only statements that may establish or falsify either side of (I1) are $2.p$ and $17.p$ (by establishing or falsifying $p@\{3..17\}$ and by updating $C[s]$), and $2.q$ and $17.q$ (by updating $C[s]$), where $q \neq p$ is any arbitrary process. Statement $2.p$ establishes both expressions, and statement $17.p$ falsifies both expressions. Statements $2.q$ and $17.q$ might potentially falsify (I1) only if executed when $p@\{3..17\} \ \wedge \ p.side = q.side = s$ holds. However, this is precluded by (I0). □

invariant $p@\{4..20\} \ \wedge \ q@\{4..20\} \ \wedge \ p \neq q \ \Rightarrow \ T = p \ \vee \ T = q$ (I2)

Proof: The only statements that may establish the antecedent are $3.p$ and $3.q$, which also establish the consequent. The only statement that may falsify the consequent is $3.r$, where r is any arbitrary process different from both p and q. However, by (I0), $r@\{3\}$ and the antecedent cannot hold simultaneously. □

invariant $p@\{9..17\} \ \wedge \ q@\{4..17\} \ \wedge \ p \neq q \ \Rightarrow \ p.rival = q \ \vee \ T = q$ (I3)

Proof: The only statements that may establish the antecedent are $8.p$ and $3.q$. Statement $8.p$ may establish the antecedent only if executed when $q@\{4..17\}$ holds. By (I0), $p@\{8\} \wedge q@\{4..17\}$ implies $p.side = 1 - q.side$, and hence, by (I1), we have $C[1 - p.side] = q$. Thus, in this case, $8.p$ also establishes $p.rival = q$. Statement $3.q$ establishes $T = q$.

The only statements that may falsify the consequent are $8.p$, $18.p$, and $3.r$, where r is any arbitrary process different from q. Statement $8.p$ preserves (I3) as shown above. The antecedent is false after the execution of $18.p$. By (I0), statement $3.r$ cannot be executed while the antecedent holds. (In particular, $r@\{3\} \wedge p@\{9..17\}$ implies $r \neq p$. But by (I0), at most two processes can be at statements 2–21.) $\Box$

invariant $p@\{5..15\} \wedge q@\{4..19\} \wedge R1[p] = p.rtoggle \wedge p \neq q \Rightarrow T = q$ (I4)

Proof: The only statements that may falsify (I4) are $4.p$ (by establishing $p@\{5..15\}$, and by updating $p.rtoggle$), $3.q$ (by establishing $q@\{4..19\}$, and by updating T), $3.r$ (by updating T), and $20.q$ and $20.r$ (by updating $R1[p]$), where r is any arbitrary process different from q. However, statement $4.p$ establishes $R1[p] \neq p.rtoggle$, and hence falsifies the antecedent. Statement $3.q$ establishes the consequent. The antecedent is false after the execution of $20.q$. By (I0), statements $3.r$ and $20.r$ cannot be executed while the antecedent holds. $\Box$

invariant $p@\{16..20\} \wedge q@\{4..17\} \wedge p \neq q \Rightarrow T = q$ (I5)

Proof: The only statements that may establish the antecedent are $8.p$, $9.p$, $14.p$, $15.p$, and $3.q$. Statement $8.p$ may establish the antecedent only if executed when $q@\{4..17\}$ holds. By (I0), $p@\{8\} \wedge q@\{4..17\}$ implies $p.side = 1 - q.side$, and hence, by (I1), we have $C[1 - p.side] = q$. Thus, in this case, $8.p$ cannot establish $p@\{16..20\}$. Statements $9.p$ and $14.p$ may establish the antecedent only if $T \neq p \wedge q@\{4..17\}$ holds, which implies $T = q$ by (I2). Statement $15.p$ may establish the antecedent only if $R1[p] = p.rtoggle \wedge q@\{4..17\}$ holds, which implies $T = q$ by (I4). Statement $3.q$ establishes the consequent.

The only statement that may falsify the consequent is $3.r$, where r is any arbitrary process different from q. Statement $3.r$ might potentially falsify (I5) only if executed when $r@\{3\}$ and the antecedent hold, which is precluded by (I0). $\Box$

invariant (Exclusion) $\big|\{p :: p@\{16\}\}\big| \leq 1$ (I6)

Proof: The only statements that may falsify (I6) are 8.p, 9.p, 14.p, and 15.p, which may falsify (I6) only if executed when $q@\{16\}$ holds for some $q \neq p$. First, consider 8.p. By (I0), $p@\{8\} \wedge q@\{16\}$ implies $p.side = 1 - q.side$, and hence, by (I1), we have $C[1 - p.side] = q$. Thus, in this case, 8.p cannot establish $p@\{16\}$.

Statements 9.p, 14.p, and 15.p might potentially falsify (I6) only if executed when $p@\{9, 14, 15\} \wedge q@\{16\}$ holds, in which case, by applying (I5) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, we have $T = p$. Thus, statements 9.p and 14.p cannot establish $p@\{16\}$. Statement 15.p may establish the antecedent only if $R1[p] = p.rtoggle$ holds, which implies $T = q$ by (I4), a contradiction of $T = p$. $\square$

invariant $p@\{6..15\} \Rightarrow R2[p] = p.rtoggle$ (I7)

invariant $p@\{8..15\} \Rightarrow Q2[p] = p.qtoggle$ (I8)

Proof: These invariants follow trivially from the structure of the algorithm. $\square$

invariant $p@\{9..15\} \wedge q@\{19, 20\} \Rightarrow q.rival = p$ (I9)

Proof: The only statements that may falsify (I9) are 8.p, 8.q, and 18.q. Statement 8.p may establish the antecedent only if executed when $q@\{19, 20\}$ holds. By (I0), $p@\{8\} \wedge q@\{19, 20\}$ implies $p.side = 1 - q.side$, and hence, by (I1), we have $C[1 - p.side] = \perp$. Thus, in this case, 8.p cannot establish $p@\{9..15\}$. The antecedent is false after the execution of 8.q. Statement 18.q may establish the antecedent only if executed when $T \neq q \wedge p@\{9..15\} \wedge q@\{18\}$ holds, which implies $T = p$ by (I2). Thus, in this case, 18.q establishes $q.rival = p$. $\square$

invariant $p@\{9..15\} \wedge q@\{20\} \Rightarrow q.temp = R2[p]$ (I10)

Proof: The only statements that may establish the antecedent are 8.p and 19.q. Statement 8.p cannot establish the antecedent as shown in the proof of (I9) above. By (I9), if 19.q establishes the antecedent, then it also establishes the consequent.

The only statements that may falsify the consequent are 5.p, 10.q, and 19.q. The antecedent is false after the execution of 5.p and 10.q. Statement 19.q preserves (I10) as shown above. $\square$

invariant $p@\{9..15\} \ \wedge\ R1[p] = \neg p.\mathit{rtoggle} \ \Rightarrow\ \big(\exists q : q \neq p :: q@\{3..20\}\big)$ (I11)

Proof: The only statements that may falsify (I11) are $4.p$ (by updating $p.\mathit{rtoggle}$), $8.p$ (by establishing $p@\{9..15\}$), and $18.q$ and $20.q$ (by falsifying $q@\{3..20\}$), where q is any arbitrary process. The antecedent is false after the execution of $4.p$. Statement $8.p$ establishes the antecedent only if $C[1 - p.\mathit{side}] = q \neq \perp$ holds for some q, in which case the consequent holds by (I1). Statement $18.q$ might potentially falsify (I11) only if executed when $p@\{9..15\} \ \wedge\ q@\{18\}$ holds. However, by applying (I5) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, we have $T = p$, and hence $18.q$ preserves $q@\{3..20\}$. Statement $20.q$ might potentially falsify (I11) only if executed when $p@\{9..15\} \ \wedge\ q@\{20\}$ holds. However, in this case, $20.q$ either preserves $q@\{20\}$ (by nonatomicity) or establishes $R1[p] = p.\mathit{rtoggle}$ by (I7), (I9), and (I10). □

invariant $p@\{10..15\} \ \wedge\ q@\{11\} \ \wedge\ p \neq q \ \Rightarrow\ q.\mathit{temp} = Q2[p] \ \vee\ T = p$ (I12)

Proof: The only statements that may establish the antecedent are $9.p$ and $10.q$. Statement $9.p$ may establish the antecedent only if $T = p$ holds. Statement $10.q$ may establish the antecedent only if executed when $p@\{10..15\} \ \wedge\ q@\{10\}$ holds, in which case, by applying (I3) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, we have $q.\mathit{rival} = p \ \vee\ T = p$. Moreover, if $q.\mathit{rival} = p$ holds, then $10.q$ establishes $q.\mathit{temp} = Q2[p]$.

The only statements that may falsify the consequent are $7.p$ (by updating $Q2[p]$), $10.q$ and $19.q$ (by updating $q.\mathit{temp}$), and $3.r$ (by updating T), where r is any arbitrary process. The antecedent is false after the execution of $7.p$ and $19.q$. Statement $10.q$ preserves (I12) as shown above. By (I0), statement $3.r$ cannot be executed while the antecedent holds. □

invariant $p@\{10..15\} \ \wedge\ q@\{12..15\} \ \wedge\ p \neq q \ \Rightarrow\ Q1[p] = p.\mathit{qtoggle} \ \vee\ T = p$ (I13)

Proof: The only statements that may establish the antecedent are $9.p$ and $11.q$. Statement $9.p$ may establish the antecedent only if $T = p$ holds. Statement $11.q$ may establish the antecedent only if executed when $p@\{10..15\} \ \wedge\ q@\{11\}$ holds. In this case, by (I3) (with 'p' $\leftarrow q$ and 'q' $\leftarrow p$), (I8), and (I12), we have $q.\mathit{rival} = p \ \vee\ T = p$, $Q2[p] = p.\mathit{qtoggle}$, and $q.\mathit{temp} = Q2[p] \ \vee\ T = p$, respectively. Thus, $11.q$ establishes the consequent in this case.

The only statements that may falsify the consequent are $6.p$, $3.r$, and $11.r$, where r is any arbitrary process. The antecedent is false after the execution of $6.p$. By (I0), statement $3.r$ cannot be executed while the antecedent holds. Finally, we consider $11.r$. If $r = p$, then statement $11.r$ $(= 11.p)$ cannot change $Q1[p]$. (Note that $p.rival$ may only hold process identifiers from the other subtree, rather than the subtree that contains p.) If $r = q$, then statement $11.r$ $(= 11.q)$ preserves (I13) as shown above. If $r \neq p$ and $r \neq q$ hold, then by (I0), $11.r$ cannot be executed while the antecedent holds. $\Box$

invariant $p@\{12, 13\} \wedge q@\{12, 13\} \wedge p \neq q \Rightarrow$
$$Q1[p] = p.qtoggle \vee Q1[q] = q.qtoggle \qquad \text{(I14)}$$

Proof: Since $T \neq p \vee T \neq q$ is always true, (I14) follows easily from (I13). $\Box$

invariant

$$\begin{array}{rll}
p@\{7..15\} \Rightarrow & p@\{7..13\} \wedge Q1[p] = \neg p.qtoggle \wedge R1[p] = \neg p.rtoggle & \text{(D1)} \\
\vee & \big(\exists q : q \neq p :: q@\{11..18\}\big) \wedge R1[p] = \neg p.rtoggle & \text{(D2)} \\
\vee & \big(\exists q : q \neq p :: q@\{19\} \wedge q.rival = p\big) \wedge R1[p] = \neg p.rtoggle & \text{(D3)} \\
\vee & \big(\exists q : q \neq p :: q@\{20\} \wedge q.rival = p \wedge q.temp = R2[p]\big) & \text{(D4)} \\
\vee & \big(\exists q : q \neq p :: q@\{2, 3, 21\}\big) \wedge R1[p] = p.rtoggle & \text{(D5)} \\
\vee & \neg\big(\exists q : q \neq p :: q@\{2..21\}\big) \wedge R1[p] = p.rtoggle & \text{(D6)} \\
\vee & p@\{7..13\} \wedge \big(\exists q : q \neq p :: q@\{20\}\big) \wedge Q1[p] = \neg p.qtoggle & \text{(D7)} \\
\vee & T \neq p & \text{(D8)}
\end{array}$$
(I15)

Proof: The only statement that may establish the antecedent is $6.p$. We consider three cases. First, if statement $6.p$ is executed while $R1[p] = \neg p.rtoggle$ holds, then it establishes (D1). Second, if $6.p$ is executed while $T \neq p$ holds, then (D8) is true after its execution. Third, if $6.p$ is executed while $R1[p] = p.rtoggle \wedge T = p$ holds, then by (I4), we have $\neg(\exists q : q \neq p :: q@\{4..19\})$, and hence, one of (D5), (D6), or $q@\{20\}$ (for some q) holds before and after the execution of $6.p$. However, if $6.p$ is executed while $q@\{20\}$ holds, then it establishes (D7).

The only statements that may falsify (D1) are $4.p$ and $6.p$ (by updating $p.qtoggle$ or $p.rtoggle$), $8.p$, $9.p$, $12.p$, and $13.p$ (by falsifying $p@\{7..13\}$), and $11.q$ and $20.q$ (by updating $Q1[p]$ or $R1[p]$), where q is any arbitrary process. The antecedent is false after the execution of $4.p$. Statement $6.p$ preserves (I15) as shown above. If statement $8.p$ or $9.p$ falsifies (D1), then it also falsifies the antecedent. Statements $12.p$ and $13.p$ cannot falsify $p@\{7..13\}$ while (D1) holds. If statement $11.q$ is executed while (D1) holds, then it establishes $q@\{11, 12\}$, and hence (D2) is established. Statement $20.q$ may falsify (D1) only if it establishes $R1[p] = p.rtoggle$. Since $20.q$ also establishes $q@\{20, 21\}$, in this case, either (D5) or (D7) is established.

The only statements that may falsify (D2) are $4.p$, $18.q$, and $20.r$, where r is any arbitrary process. The antecedent is false after the execution of $4.p$. If statement $18.q$ is executed while (D2) and the antecedent hold, then by (I0), we have $p.side = 1 - q.side$, and hence by (I1), we have $C[1 - q.side] = p$. Thus, in this case, $18.q$ establishes (D3). By (I0), $20.r$ cannot be executed while $p@\{7..15\} \land q@\{11..18\}$ holds.

The only statements that may falsify (D3) are $4.p$ (by updating $p.rtoggle$), $8.q$ and $18.q$ (by updating $q.rival$), $19.q$ (by falsifying $q@\{19\}$), and $20.r$ (by updating $R1[p]$), where r is any arbitrary process. The antecedent is false after the execution of $4.p$. Statements $8.q$ and $18.q$ cannot be executed while (D3) holds. If statement $19.q$ is executed while (D3) holds, then it establishes (D4). By (I0), $20.r$ cannot be executed while $p@\{7..15\} \land q@\{19\}$ holds.

The only statements that may falsify (D4) are $5.p$ (by updating $R2[p]$), $8.q$ and $18.q$ (by updating $q.rival$), $10.q$ and $19.q$ (by updating $q.temp$), and $20.q$ (by falsifying $q@\{20\}$). The antecedent is false after the execution of $5.p$. Statements $8.q$, $10.q$, $18.q$, and $19.q$ cannot be executed while (D4) holds. If statement $20.q$ is executed while (D4) holds, then it either preserves (D4) (by preserving $q@\{20\}$), or establishes $q@\{21\} \land R1[p] = R2[p]$. In the latter case, by (I7), it also establishes (D5).

The only statements that may falsify (D5) are $4.p$ (by updating $p.rtoggle$), $3.q$ and $21.q$ (by falsifying $q@\{2, 3, 21\}$), and $20.r$ (by updating $R1[p]$), where r is any arbitrary process. The antecedent is false after the execution of $4.p$. Statement $3.q$ establishes (D8). By (I0), if statement $21.q$ is executed while $p@\{7..15\}$ holds, then it establishes $\neg(\exists q : q \neq p :: q@\{2..21\})$. Thus, if statement $21.q$ is executed while (D5) and the antecedent hold, then it establishes (D6). By (I0), $20.r$ cannot be executed while $p@\{7..15\} \land q@\{2, 3, 21\}$ holds.

The only statements that may falsify (D6) are $4.p$, $1.q$, and $20.q$, where q is any arbitrary process different from p. The antecedent is false after the execution of $4.p$. If statement $1.q$ is executed while (D6) holds, then it establishes (D5). Statement $20.q$ cannot be executed while (D6) holds.

The only statements that may falsify (D7) are $6.p$ (by updating $p.qtoggle$), $8.p$, $9.p$, $12.p$, and $13.p$ (by falsifying $p@\{7..13\}$), $20.q$ (by falsifying $q@\{20\}$), and $11.r$ (by updating $Q1[p]$), where r is any arbitrary process. Statement $6.p$ preserves (I15) as shown in the first paragraph of this proof. If statement $8.p$ or $9.p$ falsifies (D7), then it also falsifies the antecedent. Statement $12.p$ cannot falsify $p@\{7..13\}$ while (D7) holds. Statement $13.p$ may falsify (D7) only if executed while $q@\{20\}$ holds. However, $p@\{13\} \wedge q@\{20\}$ implies $q.rival = p \wedge q.temp = R2[p]$ by (I9) and (I10). Thus, $13.p$ establishes (D4) in this case.

Assume that statement $20.q$ is executed while (D7) holds. If $20.q$ establishes $R1[p] = \neg p.rtoggle$, then it also establishes (D1). On the other hand, if $20.q$ establishes $R1[p] = p.rtoggle$, then it either preserves (D7) (by maintaining $q@\{20\}$), or establishes (D5) (by establishing $q@\{21\}$). Finally, by (I0), statement $11.r$ can be executed while $p@\{7..13\} \wedge q@\{20\}$ only if $r = p$, in which case $11.r$ $(= 11.p)$ does not change $Q1[p]$. (Note that $p.rival$ may only hold process identifiers from the other subtree, rather than the subtree that contains p.)

The only statement that may falsify (D8) is $3.p$. However, the antecedent is false after its execution. $\square$

invariant $p@\{15\} \wedge T = q \wedge p \neq q \Rightarrow R1[p] = p.rtoggle \wedge q@\{4..15\}$ (I16)

Proof: The only statements that may establish the antecedent are $14.p$ and $3.q$. Statement $14.p$ may establish $p@\{15\}$ only if executed while $T = p$ holds, in which case it cannot establish the antecedent.

Statement $3.q$ may establish the antecedent only if executed while $p@\{15\} \wedge q@\{3\}$ holds. If $3.q$ is executed while $p@\{15\} \wedge q@\{3\} \wedge T = r \wedge r \neq p$ holds, then by applying (I16) with 'q' $\leftarrow r$, we have $r@\{4..15\}$. Thus, we have $p@\{15\} \wedge q@\{3\} \wedge r@\{4..15\}$, which is impossible by (I0). Therefore, assume that $3.q$ is executed while $p@\{15\} \wedge q@\{3\} \wedge T = p$ holds. In this case, by (I15), one of disjuncts (D2)–(D5) must be true. (Disjuncts (D1) and (D7) are precluded by $p@\{15\}$; (D6), by $q@\{3\}$; (D8), by $T = p$.) Moreover, by (I0), we have $\neg(\exists r : r \neq p :: r@\{2, 4..21\})$, which precludes (D2)–(D4). Thus, we have disjunct (D5). Therefore, statement $3.q$ establishes the consequent.

The only statements that may falsify the consequent are $4.p$ (by updating $p.rtoggle$), $8.q$, $9.q$, $14.q$, and $15.q$ (by falsifying $q@\{4..15\}$), and $20.r$ (by updating $R1[p]$), where r is any arbitrary process. The antecedent is false after the execution of $4.p$. Statement $8.q$ might potentially falsify (I16) only if executed when $p@\{15\}$ holds. By (I0), $p@\{15\} \wedge q@\{8\}$ implies $p.side = 1-q.side$, and hence, by (I1), we have $C[1-q.side] = p$. Thus, in this case, $8.q$ cannot falsify $q@\{4..15\}$. Statements $9.q$ and $14.q$ cannot falsify the consequent while the antecedent holds. Statement $15.q$ might potentially falsify

(I16) only if executed when $p@\{15\} \land T = q \land R1[q] = q.rtoggle$ holds. However, this is impossible, by applying (I4) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$. By (I0), statement $20.r$ cannot be executed while $p@\{15\} \land q@\{4..15\}$ holds. □

We now prove the following *unless* properties. (A *unless* B is true if and only if the following holds: if A holds before some statement execution, then $A \lor B$ holds after that execution. Informally, A is not falsified until B is established.)

$$p@\{12..15\} \land R1[p] = p.rtoggle \land (\exists q : q \neq p :: q@\{21\}) \quad \textit{unless} \quad p@\{16\} \lor \neg(\exists q : q \neq p :: q@\{2..21\}) \tag{U1}$$

$$p@\{12..15\} \land R1[p] = p.rtoggle \land \neg(\exists q : q \neq p :: q@\{2..21\}) \quad \textit{unless} \quad p@\{16\} \lor (\exists q : q \neq p :: q@\{2,3\}) \tag{U2}$$

$$p@\{12..15\} \land R1[p] = p.rtoggle \land q@\{2,3\} \quad \textit{unless} \quad p@\{16\} \lor (q@\{4..15\} \land T = q) \tag{U3}$$

$$p@\{12..15\} \land R1[p] = p.rtoggle \land q@\{4..15\} \land T = q \quad \textit{unless} \quad p@\{16\} \tag{U4}$$

Proof: Our proof obligation is to show that, for each of (U1)–(U4), if its left-hand side is falsified, then its right-hand side is established. The only statements that may falsify $p@\{12..15\}$ are $14.p$ and $15.p$. If they falsify $p@\{12..15\}$, then they establish $p@\{16\}$. The only statements that may falsify $R1[p] = p.rtoggle$ are $4.p$ and $20.q$, where q is any arbitrary process. Statement $4.p$ cannot be executed while $p@\{12..15\}$ holds. By (I0), statement $20.q$ cannot be executed while the left-hand side of any of (U1)–(U4) holds.

For each of (U1)–(U3), it is obvious that each statement by q either preserves its left-hand side or establishes its right-hand side. (Note that (I0) and the left-hand side of (U1) together imply $\neg(\exists r : r \notin \{p, q\} :: r@\{2..21\})$.

We now consider (U4). The only other statements that might potentially falsify the left-hand side are $8.q$, $9.q$, $14.q$, $15.q$, and $3.r$, where r is any arbitrary process. We now claim that these statements cannot in fact falsify the left-hand side.

If statement $8.q$ is executed while $p@\{12..15\}$ holds, then by (I0), we have $p.side = 1 - q.side$, and hence, by (I1), we have $C[1 - q.side] = p$. Thus, in this case, $8.q$ cannot falsify $q@\{4..15\}$. Statements $9.q$ and $14.q$ cannot falsify $q@\{4..15\}$ while $T = q$ holds. Statement $15.q$ might potentially falsify the left-hand side only if executed when $R1[q] = q.rtoggle$ holds. However, by applying (I4) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, it follows that the left-hand side (U4) and $R1[q] = q.rtoggle$ cannot hold simultaneously. By (I0), $3.r$ cannot be executed while $p@\{12..15\} \ \wedge \ q@\{4..15\}$ holds. $\square$

Proof of starvation-freedom. We begin by proving livelock-freedom. It suffices to consider the loops represented by the **await** statements at statements 12–13 and statement 15. If only one process p executes one of these loops while all other processes remain in their noncritical sections or lower in the tree, then (I11) ensures $R1[p] = p.rtoggle$, and hence both loops eventually terminate.

We now consider two processes p and q, and assume $p@\{12, 13, 15\}$ and $q@\{12, 13, 15\}$. We show that either p or q eventually terminates its **await** statement. Without loss of generality, it suffices to consider the following three cases.

First, assume $p@\{12, 13\}$ and $q@\{12, 13\}$. By (I14), it follows that either p or q eventually terminates its **await** statement.

Second, assume $p@\{12, 13\}$ and $q@\{15\}$. By (I2), we have either $T = p$ or $T = q$. If $T = q$, then by (I13), we have $Q1[p] = p.qtoggle$, and hence p eventually terminates its **await** statement. On the other hand, if $T = p$, then by applying (I16) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, we have $R1[q] = q.rtoggle$, and hence q eventually terminates its **await** statement.

Third, assume $p@\{15\}$ and $q@\{15\}$. By (I2) again, we have either $T = p$ or $T = q$. Thus, by (I16), we have either $R1[p] = p.rtoggle$ or $R1[q] = q.rtoggle$, and hence either p or q eventually terminates its **await** statement.[2]

It follows that ALGORITHM NA is livelock-free. We now show that ALGORITHM NA is also starvation-free. For the sake of contradiction, assume that process p remains forever at statements 12–13 or 15. (That is, $p@\{12, 13, 15\}$ holds indefinitely.) Because of livelock-freedom, this may happen only if other processes repeatedly enter and exit their critical sections. Thus, eventually some process $q \neq p$ executes statement 18. By (I5) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, q finds $T = p$ at statement 18, and hence it executes statements 19 and 20. Moreover, by (I7), (I9), and (I10), $20.q$ eventually establishes $R1[p] = p.rtoggle \wedge q@\{21\}$, which equals the left-hand side of (U1). By applying (U1)–(U4), it follows that $R1[p] = p.rtoggle$ holds continuously until $p@\{16\}$ is established, which contradicts our assumption that $p@\{12, 13, 15\}$ holds indefinitely. It follows that ALGORITHM NA is starvation-free.

[2]In fact, with several more invariants, it can be shown that $p@\{15\} \ \wedge \ q@\{15\}$ is impossible.

APPENDIX D

DETAILED PROOF OF THE TIME-COMPLEXITY LOWER BOUND PRESENTED IN CHAPTER 7

In this appendix, our time-complexity lower bound for nonatomic systems, stated in Theorem 7.2, is proved in detail. Throughout this appendix, we use the definitions stated in Section 7.2, and assume the existence of a fixed nonatomic one-shot mutual exclusion system $\mathcal{S} = (C, P, V)$. First, we state some properties that directly follow from the definition of a regular computation.

- For each m and v, $AW_v^m(H)$ and $CW_v^m(H)$ are disjoint (see Figure 7.6). (D.1)
- For each m and v, $\mathrm{Act}(H)$ and $CW_v^m(H)$ are disjoint. (D.2)
- For each m and v, $\mathrm{Act}(H)$ and $RW_v^m(H)$ are disjoint. (D.3)
- The sets $RW_v^m(H)$ are mutually disjoint. (D.4)
- Each invocation event e_p in H is the last event in $H \mid p$. (D.5)
- For each invocation event e_p in H, $p \in \mathrm{Cvr}(H)$ holds. (D.6)
- Each atomic write or read event f_q in H is contained in some active segment $S(q, m)$. (D.7)

We now present several lemmas. Lemma D.1 asserts that information flow does not happen in a regular computation.

Lemma D.1 Consider a regular computation H in C, an event e_p in H, and a variable v. Denote H as $F \circ \langle e_p \rangle \circ G$, where F and G are subcomputations of H. If e_p reads v, then the following holds:

$$writer(v, F) = p \ \vee \ writer(v, F) = \perp \ \vee \ value(v, F) = \star.$$

Proof: Let $f_q = \mathit{writer\_event}(v, F)$. If we have either $q = p$ or $q = \perp$, then we are done. Thus, assume $q \neq p \land q \neq \perp$. By the Atomicity property, f_q is either an atomic write event of v or an invocation event on v. If f_q is an invocation event, then we have $\mathit{value}(v, F) = \star$, and hence we are done.

We claim that f_q cannot be an atomic write event. For the sake of contradiction, assume otherwise. Then, by (D.7),

- f_q is contained in solo segment $S(q, m)$, for some m. (D.8)

Thus,

- $\mathit{ce}(q, j)$ is a write to v, for some $j \leq m$; (D.9)
- $q \in AW_v^j$. (D.10)

By (D.8), H contains $S(q, m)$. Thus, by (7.11),

$$q \in \mathrm{Act}^m(H). \tag{D.11}$$

Since e_p reads v, by (D.7), and from the fact that e_p comes after f_p in H, it follows that

- e_p is contained in $S(p, l)$, for some $l \geq m$. (D.12)

Therefore, from the structure of H^l, given in (7.11), we have $p \in \mathrm{Act}^l(H)$, and hence, since $\mathrm{Act}^l(H) \subseteq \mathrm{Act}^m(H)$ (by (7.8) and (7.9)), we also have

$$p \in \mathrm{Act}^m(H). \tag{D.13}$$

Also, since e_p reads v,

- $\mathit{ce}(p, k)$ is a read of v, for some $k \leq l$. (D.14)

We consider two cases. (Note that j is defined in (D.9).)

First, if CW_v^j is nonempty, then by R2, (D.9), and (D.11), $C(q, m)$ contains an invocation event g on v. Thus, by (D.8) and (D.12), and since e_p comes after f_q, H can be written as

$$H = \cdots \circ S(q, m) \circ C(q, m) \circ \cdots \circ S(p, l) \circ \cdots ,$$

where f_q, g, and e_p are contained in $S(q, m)$, $C(q, m)$, and $S(p, l)$, respectively. Since $H = F \circ \langle e_p \rangle \circ G$, F contains $S(q, m) \circ C(q, m)$. But then F contains an event that writes v (namely, g) *after* f_q, which contradicts $f_q = \mathit{writer\_event}(v, F)$.

Second, if CW_v^j is empty, then by (D.10), (D.14), and applying R4 with 'p' $\leftarrow q$, 'm' $\leftarrow j$, and 'j' $\leftarrow k$, we have $p \in \mathrm{Cvr}^j(H)$. Since $j \leq m$ (by (D.9)), we also have $\mathrm{Cvr}^j(H) \subseteq \mathrm{Cvr}^m(H)$ (by (7.8)), and hence $p \in \mathrm{Cvr}^m(H)$. However, since $\mathrm{Cvr}^m(H)$ and $\mathrm{Act}^m(H)$ are disjoint (by (7.9)), we have a contradiction of (D.13). $\square$

We now define two "operators" on regular computations, which are used to implement the erasing strategy. Informally, the operator $erase_p$ erases all events by p from a regular computation. Toward this goal, we erase all active segments $S(p, m)$ (for each m), and also erase the corresponding covering segments $C(p, m)$, since they are no longer needed. (Thus, deployed processes that execute their invocation events in $C(p, m)$ now become reserve processes.) This operation is allowed only if p is either an active process or a reserve process. Otherwise, p is deployed to cover some variable v, and hence we cannot apply $erase_p$ directly, since that may cause q's write to v to be "uncovered" and create information flow. In that case, we first apply operator $exchange_{pq}$. Informally, $exchange_{pq}$ is an operator that exchanges the role of a deployed process p and a reserve process q, if both belong to the same set CW_v^m (see Figure 7.7). Thus, p becomes a reserve process in $exchange_{pq}(H)$, and hence can be safely erased by applying $erase_p$. (This "erase after exchange" strategy is formally described in Lemma D.6, given later in this section.)

Definition: Consider a regular computation H in C and a process p. Assume that either $p \in \mathrm{Act}(H)$ or $p \in RW_v^m(H)$ holds for some m and v. If $p \in \mathrm{Act}(H)$, then H contains solo segment $S(p, j)$ and covering segment $C(p, j)$ for each segment index j ($1 \leq j \leq m_H$). On the other hand, if $p \in RW_v^m(H)$ holds, then H contains solo segment $S(p, j)$ and covering segment $C(p, j)$ for each segment index j ($1 \leq j < m$). (See Figure 7.8; formally, this property follows from (7.5), (7.8), (7.9), (7.11), (7.13).)

We define $erase_p(H)$ to be the computation where these segments are erased, *i.e.*, $erase_p(H) = H - (S(p, 1) \circ C(p, 1)) - (S(p, 2) \circ C(p, 2)) - \cdots - (S(p, m') \circ C(p, m'))$, where m' is defined to be m_H (if p is active) or $m - 1$ (if $p \in RW_v^m(H)$ holds). $\square$

Definition: Consider a regular computation H in C, and two processes p and q. Assume that $\{p, q\} \subseteq CW_v^m(H)$, and that p is deployed to cover some solo segment $S(r, m')$ while q is not deployed in H (*i.e.*, $q \in RW_v^m(H)$). Thus, we can write H as $F \circ \langle ie(p, m)\rangle \circ G$, where $ie(p, m)$ is the invocation event by p on v, contained in the covering segment $C(r, m')$.

We define the exchange operator $\mathit{exchange}_{pq}$ to be the computation obtained by replacing $\mathit{ie}(p, m)$ with $\mathit{ie}(q, m)$, *i.e.*, $\mathit{exchange}_{pq}(H) = F \circ \langle \mathit{ie}(q, m)\rangle \circ G$. $\square$

Note that these two operators also change the relevant sets of processes. For example, if p and q are defined as in the definition of $\mathit{exchange}_{pq}$, then we have $p \in RW_v^m(H')$ and $\mathit{cp}(q) = r$, where $H' = \mathit{exchange}_{pq}(H)$.

We claim that these two operators indeed produce valid computations, and that they preserve regularity and the structure of H (*e.g.*, $\mathrm{Act}(H)$, $\mathrm{Cvr}(H)$, *etc.*), with appropriate changes. This claim is formalized in Lemmas D.2 and D.3.

Lemma D.2 Consider a regular computation H in C with induction number m_H, and two processes p and q. Assume that $\{p, q\} \subseteq CW_v^m(H)$, p is deployed to cover some solo segment $S(r, m')$, and that q is not deployed in H (*i.e.*, $q \in RW_v^m(H)$). Define $H' = \mathit{exchange}_{pq}(H)$. Then, H' is a regular computation in C with induction number m_H, satisfying the following for each j ($1 \le j \le m_H$), k ($1 \le k \le m_H$), variable w, and process s:

$$P(H') = P(H); \tag{D.15}$$

$$\mathrm{Act}^j(H') = \mathrm{Act}^j(H); \tag{D.16}$$

$$\mathrm{Cvr}^j(H') = \mathrm{Cvr}^j(H); \tag{D.17}$$

$$AW_w^j(H') = AW_w^j(H); \tag{D.18}$$

$$CW_w^j(H') = CW_w^j(H); \tag{D.19}$$

$$RW_w^j(H') = \begin{cases} (RW_w^j(H) \cup \{p\}) - \{q\}, & \text{if } j = m \text{ and } w = v \\ RW_w^j(H), & \text{otherwise.} \end{cases} \tag{D.20}$$

Proof: Note that, by the definition of $\mathit{exchange}_{pq}$, the only difference between H and H' is that $\mathit{ie}(p, m)$ is replaced by $\mathit{ie}(q, m)$. Moreover, both $\mathit{ie}(p, m)$ and $\mathit{ie}(q, m)$ are invocation events on the same variable v. It follows that processes other than p or q cannot distinguish between H' and H, and hence we have $H' \in C$.

Assertions (D.15)–(D.20) follow immediately from the definition of $\mathit{exchange}_{pq}$. Since H satisfies (7.3)–(7.13) and R1–R4, by applying (D.15)–(D.20), assertions (7.3)–(7.13) and R1–R4 follow immediately. It follows that H' is also regular. $\square$

Lemma D.3 Consider a regular computation H in C with induction number m_H, and a process p. Assume either of the following:

- $p \in \text{Act}(H)$, or **(A)**
- $p \in RW_v^m(H) \;\wedge\; (AW_v^m(H) = \{\} \;\vee\; |CW_v^m(H)| \geq 2)$, for some m and v. **(B)**

Define $H' = \mathit{erase}_p(H)$. Then, H' is a regular computation in C with induction number m_H, satisfying the following for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q:

$$P(H') \;=\; P(H) - \{p\}; \tag{D.21}$$

$$\text{Act}^j(H') \;=\; \text{Act}^j(H) - \{p\}; \tag{D.22}$$

$$\text{Cvr}^j(H') \;=\; \text{Cvr}^j(H) - \{p\}; \tag{D.23}$$

$$AW_w^j(H') \;=\; AW_w^j(H) - \{p\}; \tag{D.24}$$

$$CW_w^j(H') \;=\; CW_w^j(H) - \{p\}; \tag{D.25}$$

$$RW_w^j(H') \;\supseteq\; RW_w^j(H). \tag{D.26}$$

Proof: Step 1. First, we show that H' is a valid computation in C.

Note that operator erase_p completely removes events by p, plus p's covering segments (which consist only of invocation events). By (D.5), each such removed event (except events by p) is the last event by that process. Therefore,

- for each process q such that $H' \mid q \neq \langle\rangle$, $H' \mid q$ is either $H \mid q$, or the subcomputation of $H \mid q$ obtained by removing its last event. (D.27)

We use induction on the length of H', and inductively apply P2$'$ to each event in H' in order. Consider an event e_q in H'. By the definition of erase_p, e_q is also an event of H. Denote H and H' as

$$H = F \circ \langle e_q \rangle \circ G \quad \text{and} \quad H' = F' \circ \langle e_q \rangle \circ G', \tag{D.28}$$

where F and G (respectively, F' and G') are subcomputations of H (respectively, H'). Also, by the definition of erase_p, and by (D.27), we have the following:

- F' is a subcomputation of F; (D.29)
- $F' \mid q = F \mid q$. (D.30)

By induction, we have

$$F' \in C. \tag{D.31}$$

Our proof obligation is to show $F' \circ \langle e_q \rangle \in C$. We now establish the following claim.

Claim 1: If e_q reads a variable u, and if $\mathit{writer\_event}(u, F') \neq \perp$, then $\mathit{value}(u, F') = \mathit{value}(u, F)$ holds.

Proof of Claim: By (D.7),

- e_q is contained in some solo segment $S(q, l)$. (D.32)

Hence, since e_q reads u,

- $ce(q, j)$ reads u, for some $j \leq l$. (D.33)

Also, by (7.11) and (D.32), we have

$$q \in \mathrm{Act}^{l}(H). \tag{D.34}$$

Note that, by (D.29), $\mathit{writer\_event}(u, F') \neq \perp$ implies

$$\mathit{writer}(u, F) \neq \perp. \tag{D.35}$$

We consider two cases. First, assume that $\mathit{writer}(u, F) = q$ holds. Let $f_q = \mathit{writer\_event}(u, F)$. By (D.30), f_q is also contained in F'. By (D.29), this implies that $f_q = \mathit{writer\_event}(u, F')$, and hence the claim follows.

Second, assume that $\mathit{writer}(u, F) \neq q$ holds. In this case, by (D.35), and by applying Lemma D.1 with 'e_p' $\leftarrow e_q$ and 'v' $\leftarrow u$, we have $\mathit{value}(u, F) = \star$. (The assumptions stated in Lemma D.1 follow from (D.28) and the assumption of Claim 1.) If we also have $\mathit{value}(u, F') = \star$, then we are done. For the sake of contradiction, assume otherwise. Define f_r as

$$f_r = \mathit{writer\_event}(u, F'). \tag{D.36}$$

(By the assumption of Claim 1, $f_r \neq \perp$ holds.) Since $\mathit{writer}(u, F) \neq q$, we have $r \neq q$. Since f_r writes to u a value different from $\star$, by the Atomicity property, f_r is an atomic write event on u, and hence, by (D.7),

- f_r is contained in some solo segment $S(r, l')$. (D.37)

Hence, by (7.11),

$$r \in \mathrm{Act}^{l'}(H). \tag{D.38}$$

Since f_r is contained in F', by (D.28), (D.29), and the definition of erase_p,

- f_r precedes e_q in H, and (D.39)
- $r \neq p$. (D.40)

Combining (D.32), (D.37), and (D.39), we also have $l' \leq l$. Since f_r writes u,

- $ce(r, k)$ is a write to u for some $k \leq l'$. (D.41)

We claim that $CW_u^k(H)$ is nonempty. Assume otherwise. Then, by (D.33) and (D.41), and by applying R4 with 'm' $\leftarrow k$, 'p' $\leftarrow r$, and 'v' $\leftarrow u$, we have $q \in \mathrm{Cvr}^k(H)$. Since $k \leq l$ (by (D.41)), we also have $\mathrm{Cvr}^k(H) \subseteq \mathrm{Cvr}^l(H)$ (by (7.8)), and hence we have $q \in \mathrm{Cvr}^l(H)$. However, since $\mathrm{Cvr}^l(H)$ and $\mathrm{Act}^l(H)$ are disjoint (by (7.9)), we have a contradiction of (D.34).

It follows that $CW_u^k(H)$ is nonempty, and hence, by (D.38) and (D.41), and by applying R2 with 'p' $\leftarrow r$, 'm' $\leftarrow l'$, and 'j' $\leftarrow k$, it follows that $C(r, l'; H)$ contains an invocation event g on u. Thus, by (D.32), (D.37), and (D.39), H can be written as

$$H = \cdots \circ S(r, l') \circ C(r, l'; H) \circ \cdots \circ S(q, l) \circ \cdots ,$$

where f_r, g, and e_q are contained in $S(r, l')$, $C(r, l'; H)$, and $S(q, l)$, respectively. By (D.40), and by the definition of erase_p, $C(r, l'; H)$ is also contained in F'. But then F' contains an event that writes u (namely, g) *after* f_r, which contradicts (D.36). □

The following claim establishes $F' \circ \langle e_q \rangle \in C$, and hence, by induction, $H' \in C$.

Claim 2: $F' \circ \langle e_q \rangle \in C$.

Proof of Claim: We consider three cases. First, if e_q is *not* a read event, then by (D.31), and by applying P2$'$ with 'H' $\leftarrow F$, 'e_p' $\leftarrow e_q$, and 'G' $\leftarrow F'$, the claim follows.

Second, assume that e_q reads some variable u, and that $\mathit{writer\_event}(u, F') \neq \bot$ holds. By Claim 1, we again have $\mathit{value}(u, F') = \mathit{value}(u, F)$. Thus, by (D.31), and by applying P2$'$ with 'H' $\leftarrow F$, 'e_p' $\leftarrow e_q$, 'G' $\leftarrow F'$, and 'v' $\leftarrow u$, the claim follows.

Third, assume that e_q reads some variable u, and that $\mathit{writer\_event}(u, F') = \perp$ holds. By (D.7), e_q is contained in some solo segment $S(q, m)$. Hence, by the definition of a solo segment, $(F' \mid q) \circ \langle e_q \rangle$ is a valid solo computation, that is,

$$(F' \mid q) \circ \langle e_q \rangle \in C. \tag{D.42}$$

Since $F' \mid q$ is a subcomputation of F', we also have

$$\mathit{value}(u, F' \mid q) = \mathit{value}(u, F') = (\text{initial value of } u). \tag{D.43}$$

By (D.31), (D.42), (D.43), and applying P2$'$ with 'H' $\leftarrow F' \mid q$, 'e_p' $\leftarrow e_q$, 'G' $\leftarrow F'$, and 'v' $\leftarrow u$, the claim follows. $\square$

Step 2. We now show that H' is regular, and that H' satisfies (D.21)–(D.26). Assertions (D.21)–(D.26) follow immediately from the definition of erase_p. Since H satisfies (7.3)–(7.13), by applying (D.21)–(D.26), assertions (7.3)–(7.13) follow easily. We now show that H' satisfies each of R1–R4.

In order to show that H' satisfies R1, consider an event e_q contained in a covering segment $C(r, l; H')$. By the definition of erase_p, we have $C(r, l; H') = C(r, l; H)$, $q \neq p$, and $r \neq p$. Thus, by applying R1 to e_q in H, we have the following for some $j \leq l$ and variable u: $e_r = \mathit{ie}(q, j)$, $q \in CW_u^j(H)$, $\mathit{cp}(q; H) = r$, and $\mathit{ce}(r, j)$ is a write to u. By (D.25) and $q \neq p$, we have $q \in CW_u^j(H')$. By $r \neq p$, and by the definition of erase_p, $\mathit{cp}(q; H') = r$ also holds. Thus, we have R1.

In order to show that H' satisfies R2, consider a process $q \in \mathrm{Act}^l(H')$ and a segment index $j \leq l$, such that $\mathit{ce}(q, j)$ writes some variable u and $CW_u^j(H')$ is nonempty. By (D.22), we have $q \in \mathrm{Act}^l(H)$ and $q \neq p$. By (D.25), $CW_u^j(H)$ is also nonempty. Thus, by applying R2 to q in H, it follows that $C(q, l; H)$ has exactly one invocation event on u (plus perhaps some other events), which is $\mathit{ie}(r, j)$ for some $r \in CW_u^j(H)$. As shown above, since $q \neq p$, we have $C(q, l; H') = C(q, l; H)$. Moreover, since r is deployed to cover q, we have $r \neq p$. (Note that Conditions (A) and (B) in the statement of the lemma imply that p is not deployed in H.) Thus, by (D.25), we have $r \in CW_u^j(H')$. It follows that $C(q, l; H')$ contains exactly one invocation event on u (namely, $\mathit{ie}(r, j)$), where $r \in CW_u^j(H')$. Thus, we have R2.

Since H' is a subcomputation of H, R3 follow easily.

Before showing that H' satisfies R4, we need the following claim.

Claim 3: If both $AW_w^j(H)$ and $CW_w^j(H)$ are nonempty, then so is $CW_w^j(H')$.

Proof of Claim: We consider two cases. First, if Condition (A) holds, then since $\mathrm{Act}(H)$ and $CW_w^j(H)$ are disjoint (by (D.2)), we have $p \notin CW_w^j(H)$. Therefore, by (D.25), $CW_w^j(H')$ equals $CW_w^j(H)$, and hence is nonempty.

Second, assume that Condition (B) holds. If $(j, w) = (m, v)$, then since $AW_w^j(H)$ is nonempty, we have $|CW_v^m(H)| \geq 2$, and hence, by (D.25), $CW_w^j(H')$ $(= CW_v^m(H'))$ is nonempty. On the other hand, if $(j, w) \neq (m, v)$, then $CW_w^j(H)$ and $CW_v^m(H)$ are disjoint by (7.5). Moreover, by Condition (B), we have $p \in RW_v^m(H) \subseteq CW_v^m(H)$. Thus we have $p \notin CW_w^j(H)$. Therefore, by (D.25), we have $CW_w^j(H') = CW_w^j(H)$, and hence $CW_w^j(H')$ is nonempty. $\square$

We now claim that H' satisfies R4. Consider some segment index l, process q, and variable u such that $q \in AW_u^l(H')$ and $CW_u^l(H')$ is empty. By (D.24), we have $q \in AW_u^l(H)$ and $q \neq p$. By Claim 3, $CW_u^l(H)$ is also empty. Therefore, by applying R4 to H with 'm' $\leftarrow l$, 'p' $\leftarrow q$, and 'v' $\leftarrow u$, it follows that, for each segment index j and each process $r \in \mathrm{Act}^j(H)$ different from q, the following hold:

(i) if $j < l$ and $ce(r, j)$ is a write to u, then $CW_u^j(H)$ is nonempty;

(ii) if $j < l$ and $ce(r, j)$ is a read of u, then $r \in \mathrm{Cvr}^l(H)$ holds;

(iii) if $l \leq j \leq m_H$, then $ce(r, j)$ does not access u.

By (D.22), $r \in \mathrm{Act}^j(H')$ implies $r \in \mathrm{Act}^j(H)$ and $r \neq p$. Note that, if (i) is true, then $r \in AW_u^j(H)$ holds by definition. Thus, by Claim 3, it follows that $CW_u^j(H')$ is also nonempty. Also note that, since $r \neq p$, if (ii) is true, then $r \in \mathrm{Cvr}^l(H)$ and (D.23) imply $r \in \mathrm{Cvr}^l(H')$. From these assertions, R4 easily follows. $\square$

The next lemma is a simple application of Lemma D.3. It states that we can safely erase any subset of active processes.

Lemma D.4 Consider a regular computation H in C with induction number m_H, and a subset $K = \{p_1,\ p_2,\ \ldots,\ p_h\}$ of $\mathrm{Act}(H)$.

Define H' as the result of applying operation $erase_{p_i}$ to H for each $p_i \in K$, *i.e.*,

$$H' = erase_{p_h}(erase_{p_{h-1}}(\cdots erase_{p_2}(erase_{p_1}(H)) \cdots)).$$

Then, H' is a regular computation in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H') \leq \pi_{\max}(H)$; (D.44)
- for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q,

$$\begin{aligned} P(H') &= P(H) - K; && \text{(D.45)}\\ \mathrm{Act}^j(H') &= \mathrm{Act}^j(H) - K; && \text{(D.46)}\\ \mathrm{Cvr}^j(H') &= \mathrm{Cvr}^j(H); && \text{(D.47)}\\ AW^j_w(H') &= AW^j_w(H) - K; && \text{(D.48)}\\ CW^j_w(H') &= CW^j_w(H); && \text{(D.49)}\\ RW^j_w(H') &\supseteq RW^j_w(H). && \text{(D.50)} \end{aligned}$$

Proof: By inductively applying Lemma D.3, assertions (D.45)–(D.50) follow easily. (As for (D.47) and (D.49), note that $K \subseteq \mathrm{Act}(H)$ implies $K \cap \mathrm{Cvr}^j(H) = K \cap CW^j_w(H) = \{\}$, for each j and w.)

We now prove that H' satisfies (D.44). It suffices to show that, for each covering pair (j, w) in H', we have $\pi(j, w; H') \leq \pi(j, w; H)$. Consider each covering pair (j, w) in H'. By definition, we have

$$AW^j_w(H') \neq \{\} \ \wedge \ CW^j_w(H') \neq \{\}. \qquad \text{(D.51)}$$

By (D.48), we have

$$|AW^j_w(H')| \leq |AW^j_w(H)|. \qquad \text{(D.52)}$$

Thus, since both H and H' have induction number m_H, by the definition of 'req', given in (7.15), we have

$$\mathrm{req}(j, w; H') \leq \mathrm{req}(j, w; H). \qquad \text{(D.53)}$$

By (D.49), (D.51), and (D.52), it follows that (j, w) is also a covering pair in H. Thus, by (7.16), we have $\pi(j, w; H) = \max\{0, \ \mathrm{req}(j, w; H) - |CW^j_w(H)|\}$. Combining this with (D.49) and (D.53), assertion (D.44) follows. $\square$

The following lemma ensures that a regular computation with a "low" maximum potential has "enough" reserve writers, for each covering pair. (Recall that, by (7.16), the potential $\pi(m, v)$ of a covering pair (m, v) decreases as the number of covering processes $= |CW^m_v|$ increases.)

Lemma D.5 Consider a regular computation H in C with induction number m_H. Assume the following:

- $m_H \leq c - 2$, and (D.54)
- $\pi_{\max}(H) \leq c$. (D.55)

Then, for each covering pair (j, w) of H, we have

$$|RW_w^j| > |AW_w^j|.$$

Proof: For each covering pair (j, w), by (D.55), and by the definitions of $\pi_{\max}$ and 'req' (given in (7.15)–(7.17)), we have $\mathrm{req}(j, w; H) - |CW_w^j| \leq c$, and hence,

$$|CW_w^j| \geq \mathrm{req}(j, w; H) - c = c \cdot (|AW_w^j| + c - m_H - 1) > c \cdot |AW_w^j|, \tag{D.56}$$

where the last inequality follows from (D.54).

Note that, by R1 and R2, there exists a one-to-one correspondence between deployed processes in CW_w^j and covering segments $C(q, k)$ such that $k \geq j$ and $q \in AW_w^j \cap \mathrm{Act}^k(H)$ (see Figure 7.6). Thus,

$$\begin{aligned}
|RW_w^j| &= |CW_w^j| - \sum_{k=j}^{m_H} |AW_w^j \cap \mathrm{Act}^k(H)| \\
&\geq |CW_w^j| - (m_H - j + 1) \cdot |AW_w^j| \\
&\geq |CW_w^j| - m_H \cdot |AW_w^j| \\
&> (c - m_H) \cdot |AW_w^j| && \{\text{by (D.56)}\} \\
&> |AW_w^j|, && \{\text{by (D.54)}\}
\end{aligned}$$

which completes the proof. □

The next lemma is a simple application of Lemmas D.2 and D.3. Given a regular computation H satisfying $\pi_{\max}(H) \leq c$, and a process $p \in P(H)$, we can erase p from H as follows. If p is either an active or a reserve process, then we apply $erase_p$. On the other hand, if p is deployed, then $\pi_{\max}(H) \leq c$ implies that there exists a reserve process q that may be exchanged with p by applying $exchange_{pq}$ to H. After applying $exchange_{pq}$, p becomes a reserve process, so we can erase p by applying $erase_p$. (Note that this procedure may increase the maximum potential, by reducing the number of covering processes.)

Lemma D.6 Consider a regular computation H in C with induction number m_H, and a process p. Assume the following:

- $m_H \leq c - 2$, (D.57)
- $\pi_{\max}(H) \leq c$, and (D.58)
- $p \in P(H)$. (D.59)

Then, there exists a regular computation H' in C with induction number m_H, satisfying the following for each segment index j $(1 \leq j \leq m_H)$ and variable w:

- if $AW_w^j(H')$ is nonempty and $CW_w^j(H')$ is empty, then $CW_w^j(H)$ is also empty; (D.60)
- if $p \in CW_w^j(H)$, then $\pi(j, w; H') \leq \pi(j, w; H) + 1$; (D.61)
- if $p \notin CW_w^j(H)$, then $\pi(j, w; H') \leq \pi(j, w; H)$; (D.62)
- the following hold:

$$\begin{aligned} P(H') &= P(H) - \{p\}; && \text{(D.63)} \\ \mathrm{Act}^j(H') &= \mathrm{Act}^j(H) - \{p\}; && \text{(D.64)} \\ \mathrm{Cvr}^j(H') &= \mathrm{Cvr}^j(H) - \{p\}; && \text{(D.65)} \\ AW_w^j(H') &= AW_w^j(H) - \{p\}; && \text{(D.66)} \\ CW_w^j(H') &= CW_w^j(H) - \{p\}. && \text{(D.67)} \end{aligned}$$

Proof: First, we consider the case in which $p \in \mathrm{Act}(H)$ holds. Let $H' = erase_p(H)$. By applying Lemma D.3, assertions (D.63)–(D.67) can be easily shown to be true. Moreover, for each segment index j and variable w, $p \in \mathrm{Act}(H)$ implies $p \notin CW_w^j(H)$ (by (7.4) and (7.5)). Therefore, by (D.67), we have $CW_w^j(H') = CW_w^j(H)$, and hence (D.60) follows. Combining $CW_w^j(H') = CW_w^j(H)$ with (7.15), (7.16), and (D.66), we also have $\pi(j, w; H') \leq \pi(j, w; H)$, and hence we also have (D.61) and (D.62).

Thus, in the rest of the proof, we may assume $p \notin \mathrm{Act}(H)$. In this case, by (7.4), (7.5), and (D.59), we have the following:

$$p \in CW_v^m(H), \quad \text{for some } m \ (1 \leq m \leq m_H) \text{ and variable } v. \qquad \text{(D.68)}$$

We now establish the following simple claim.

Claim 1: $AW_v^m(H) = \{\} \ \vee \ |RW_v^m(H)| \geq 2$.

Proof of Claim: If $AW_v^m(H)$ is empty, then we are done. Otherwise, by (D.68), (m, v) is a covering pair in H. Thus, by applying Lemma D.5, we have $|RW_v^m(H)| > |AW_v^m(H)| \geq 1$, and hence the claim follows. (Assumptions (D.54) and (D.55) stated in Lemma D.5 follow from (D.57) and (D.58), respectively.) $\square$

Since $RW_v^m(H) \subseteq CW_v^m(H)$ (by (7.13)), Claim 1 implies the following:

$$AW_v^m(H) = \{\} \ \vee \ |CW_v^m(H)| \geq 2. \tag{D.69}$$

The rest of the proof consists of two steps. In Step 1, we construct a regular computation H' (in C) with induction number m_H, satisfying (D.63)–(D.67). In Step 2, we show that H' also satisfies (D.60)–(D.62).

Step 1. We consider two cases.

First, if p is *not* deployed in H, then by (D.68), we have $p \in RW_v^m(H)$. In this case, let $H' = \mathit{erase}_p(H)$. By applying Lemma D.3, it follows that H' is a regular computation in C with induction number m_H, satisfying (D.63)–(D.67). (Condition (B) of Lemma D.3 follows from $p \in RW_v^m(H)$ and (D.69).)

Second, assume that p is deployed in H to cover some solo segment $S(r, l)$ (*i.e.*, p's invocation event is contained in $C(r, l; H)$). In this case, by applying R1 with 'q' $\leftarrow p$, 'p' $\leftarrow r$, and 'm' $\leftarrow l$, it follows that $p \in CW_u^j(H)$ and $r \in AW_u^j(H)$ hold for some segment index j and variable u. By (7.5) and (D.68), we have $(j, u) = (m, v)$, and hence we have $r \in AW_v^m(H)$. Thus,

- $AW_v^m(H)$ is nonempty, (D.70)

and by Claim 1, $RW_v^m(H)$ is also nonempty. Fix a process $q \in RW_v^m(H)$, and let $H'' = \mathit{exchange}_{pq}(H)$. By applying Lemma D.2, it follows that H'' is a regular computation in C with induction number m_H, satisfying the following for for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, and variable w:

$$P(H'') = P(H); \tag{D.71}$$
$$\mathrm{Act}^j(H'') = \mathrm{Act}^j(H); \tag{D.72}$$
$$\mathrm{Cvr}^j(H'') = \mathrm{Cvr}^j(H); \tag{D.73}$$
$$AW_w^j(H'') = AW_w^j(H); \tag{D.74}$$

$$CW^j_w(H'') = CW^j_w(H); \tag{D.75}$$

$$RW^j_w(H'') = \begin{cases} (RW^j_w(H) \cup \{p\}) - \{q\}, & \text{if } j = m \text{ and } w = v \\ RW^j_w(H), & \text{otherwise.} \end{cases} \tag{D.76}$$

By (D.76), we have $p \in RW^m_v(H'')$. Also, by (D.69), (D.70), and (D.75), we have $|CW^m_v(H'')| \geq 2$. That is,

$$p \in RW^m_v(H'') \ \wedge \ |CW^m_v(H'')| \geq 2. \tag{D.77}$$

Let $H' = \mathit{erase}_p(H'')$. We now apply Lemma D.3 with 'H' $\leftarrow H''$. (Condition (B) of Lemma D.3 follows from (D.77).) It follows that H' is a regular computation in C with induction number m_H, satisfying the following for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, and variable w:

$$P(H') = P(H'') - \{p\}; \tag{D.78}$$

$$\text{Act}^j(H') = \text{Act}^j(H'') - \{p\}; \tag{D.79}$$

$$\text{Cvr}^j(H') = \text{Cvr}^j(H'') - \{p\}; \tag{D.80}$$

$$AW^j_w(H') = AW^j_w(H'') - \{p\}; \tag{D.81}$$

$$CW^j_w(H') = CW^j_w(H'') - \{p\}. \tag{D.82}$$

By combining (D.71)–(D.75) with (D.78)–(D.82), it follows that H' satisfies (D.63)–(D.67).

Step 2. We now show that H' constructed above satisfies (D.60)–(D.62). We prove (D.60) by proving its logical equivalent: if $CW^j_w(H')$ is empty and $CW^j_w(H)$ is nonempty, then $AW^j_w(H')$ is empty. By (D.67), the antecedent of this implication implies $CW^j_w(H) = \{p\}$. However, by (7.5) and (D.68), this implies $(j, w) = (m, v)$, and hence, by (D.69), it follows that $AW^j_w(H)$ is empty. Thus, by (D.66), $AW^j_w(H')$ is also empty. It follows that H' satisfies (D.60).

Since $AW^m_v(H)$ and $CW^m_v(H)$ are disjoint (by (D.1)), by (D.68), we have $p \notin AW^m_v(H)$, and hence, by (D.66), we have $AW^m_v(H') = AW^m_v(H)$. Thus, since both H and H' have induction number m_H, by the definition of 'req' (given in (7.15)), we have

$$\text{req}(m, v; H') = \text{req}(m, v; H).$$

Also, by (D.67) and (D.68), we have

$$|CW_v^m(H')| = |CW_v^m(H)| - 1.$$

Combining these two assertions with the definition of $\pi(m, v)$ (given in (7.16)), (D.61) easily follows. (Note that $p \in CW_v^m(H)$ implies $(j, w) = (m, v)$.)

In order to prove (D.62), consider a segment index j and a variable w such that $(j, w) \neq (m, v)$. If (j, w) is *not* a covering pair in H', then by the definition of π, we have $\pi(j, w; H') = 0$ and $\pi(j, w; H) \geq 0$, and hence (D.62) follows easily.

On the other hand, if (j, w) *is* a covering pair in H', then by (7.5), (D.68), and $(j, w) \neq (m, v)$, we have $p \notin CW_w^j(H)$, and hence, by (D.67), we have

$$CW_w^j(H') = CW_w^j(H).$$

Also, by (D.66), we have $|AW_w^j(H')| \leq |AW_w^j(H)|$, and hence, since both H and H' have induction number m_H, by the definition of 'req' (given in (7.15)), we have

$$\text{req}(j, w; H') \leq \text{req}(j, w; H).$$

Combining these two assertions with (7.16), assertion (D.62) easily follows. □

We now formally present the chain erasing procedure, described in Section 7.3. Here, we denote the set of processes to erase by K. The procedure is shown in Figure D.1. The following lemma proves its correctness.

Lemma D.7 Consider a regular computation H in C with induction number m_H, and a set K of processes. Assume the following:

- $m_H \leq c - 2$, (D.83)
- $\pi_{\max}(H) = 0$, and (D.84)
- $K \subseteq P(H)$. (D.85)

Then, there exists a regular computation H' in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H') < c$; (D.86)
- $|\text{Act}(H')| \geq |\text{Act}(H) - K| - |K|/(c-1)$; (D.87)
- for each segment index j ($1 \leq j \leq m_H$) and variable w,
  - if $AW_w^j(H')$ is nonempty and $CW_w^j(H')$ is empty, then $CW_w^j(H)$ is also empty; (D.88)

```
1: $F := H$;
2: for $i := 1$ to $h$ do
       /* loop invariant:
           1. $F$ is a regular computation in $C$ with induction number $m_H$;
           2. $\pi_{\max}(F) < c$
       */
3:     if $p_i \in P(F)$ then
4:         erase $p_i$ from $F$ by applying Lemma D.6 with '$H$' $\leftarrow F$ and '$p$' $\leftarrow p_i$;
           let the resulting computation be $F$;
5:         while $\pi_{\max}(F) = c$ do
               /* loop invariant: there exists exactly one covering pair $(m, v)$
                    satisfying $\pi(m, v; F) = c$ */
6:             choose a process $r$ from $AW_v^m(F)$;
7:             erase $r$ from $F$ by applying Lemma D.6 with '$H$' $\leftarrow F$ and '$p$' $\leftarrow r$;
               let the resulting computation be $F$
   od fi od
8: $H' := F$
```

Figure D.1: The chain-erasing procedure to erase processes in $K = \{p_1,\ p_2,\ \ldots,\ p_h\}$. We assume that H is a regular computation with $\pi_{\max}(H) = 0$, and that $K \in P(H)$ holds. The correctness of this algorithm is formally proved in Lemma D.7.

– the following hold:

$$P(H') \subseteq P(H) - K; \tag{D.89}$$

$$\mathrm{Act}^j(H') \subseteq \mathrm{Act}^j(H) - K; \tag{D.90}$$

$$\mathrm{Cvr}^j(H') \subseteq \mathrm{Cvr}^j(H) - K; \tag{D.91}$$

$$AW_w^j(H') \subseteq AW_w^j(H) - K; \tag{D.92}$$

$$CW_w^j(H') \subseteq CW_w^j(H) - K. \tag{D.93}$$

Proof: Arbitrarily index processes in K as $K = \{p_1,\ p_2,\ \ldots,\ p_h\}$, where $h = |K|$. We prove the lemma by applying the algorithm shown in Figure D.1. We claim that the algorithm preserves the following four invariants after executing line 1.

invariant F is a regular computation in C with induction number m_H. (I1)

invariant For each segment index j and variable w, if $AW_w^j(F)$ is nonempty and $CW_w^j(F)$ is empty, then $CW_w^j(H)$ is also empty. (I2)

invariant $\pi_{\max}(F) \le c$. (I3)

(Note that (I3) is weaker than the loop invariant $\pi_{\max}(F) < c$ stated in Figure D.1, because (I3) holds throughout lines 2–8.) It is easy to see that line 1 establishes these invariants. In particular, (I3) follows from (D.84); the other invariants are trivial. We now show that, for each line s $(2 \le s \le 7)$ of the algorithm, if line s is executed while invariants (I1)–(I3) hold, then they also hold after the execution of line s. This will establish each of (I1)–(I3) as an invariant. Since F may be updated only by execution of lines 4 and 7, it suffices to check these two lines to prove the correctness of (I1)–(I3).

First, we claim that we can apply Lemma D.6 at these lines. It suffices to show that the assumptions (D.57)–(D.59) stated in Lemma D.6 are satisfied before executing either line 4 or 7. Assumptions (D.57) and (D.58) follow from (D.83) and (I3), respectively; (D.59) is guaranteed by lines 3 and 6.

We now claim that execution of these lines preserves invariants (I1)–(I3). Let F_{old} be the value of F before executing line 4 or 7, and F_{new} be the value of F after executing that line. Lemma D.6 implies that (I1) is preserved. (That is, if F_{old} satisfies (I1), then so does F_{new}.)

Also, by applying (D.60) and (D.66) with 'H' $\leftarrow F_{\text{old}}$ and 'H'' $\leftarrow F_{\text{new}}$, we have the following for each segment index j and variable w.

- If $AW_w^j(F_{\text{new}})$ is nonempty and $CW_w^j(F_{\text{new}})$ is empty, then $CW_w^j(F_{\text{old}})$ is also empty. (D.94)
- $AW_w^j(F_{\text{new}}) \subseteq AW_w^j(F_{\text{old}})$ holds. In particular, if $AW_w^j(F_{\text{new}})$ is nonempty, then so is $AW_w^j(F_{\text{old}})$. (D.95)

By (D.94) and (D.95), it follows that lines 4 and 7 preserve (I2). In particular, if $AW_w^j(F_{\text{new}})$ is nonempty and $CW_w^j(F_{\text{new}})$ is empty, then by (D.94) and (D.95), it follows that $AW_w^j(F_{\text{old}})$ is nonempty and $CW_w^j(F_{\text{old}})$ is empty. Since F_{old} satisfies (I2), this in turn implies that $CW_w^j(H)$ is empty.

In order to show that (I3) is an invariant, we need to prove the following assertions.

- The execution of line 4 or 7 may increase $\pi(j, w; F)$ for at most one pair (j, w) (where $1 \le j \le m_H$ and $w \in V$). Moreover, if such a pair exists, then $p \in CW_w^j(F_{\text{old}}) \wedge \pi(j, w; F_{\text{new}}) = \pi(j, w; F_{\text{old}}) + 1$ holds. (D.96)
- Line 4 is executed only if $\pi_{\max}(F_{\text{old}}) < c$ holds. (D.97)
- If line 4 establishes $\pi_{\max}(F_{\text{new}}) = c$, then it also establishes the following: there exists exactly one covering pair (m, v) of F_{new} that satisfies $\pi(m, v; F_{\text{new}}) = c$. (D.98)
- Assume that lines 6 and 7 are executed when $\pi_{\max}(F_{\text{old}}) = c$ holds and that there exists exactly one covering pair (m, v) of F_{old} satisfying $\pi(m, v; F_{\text{old}}) = c$. In this case, line 7 establishes the following:
  - $\pi(m, v; F_{\text{new}}) = 0,$ (D.99)

– there exists at most one covering pair (m', v') in F_{new} that satisfies $\pi(m', v'; F_{\text{new}}) = c$, and (D.100)

– $\pi_{\max}(F_{\text{new}}) \leq c$. (D.101)

Proof of (D.96)–(D.101)**:** Assertion (D.96) easily follows from applying (D.61) and (D.62) with 'H' $\leftarrow F_{\text{old}}$ and 'H'' $\leftarrow F_{\text{new}}$, and from the fact that $p \in CW^j_w(F_{\text{old}})$ holds for at most one pair (j, w), namely, $(ci(p), cv(p))$.

Assertion (D.97) follows easily by inspecting the algorithm. In particular, line 1 establishes $\pi_{\max}(F) = 0$ by (D.84), and the **while** loop of lines 5–7 establishes $\pi_{\max}(F) < c$ upon termination.

We now prove (D.98). By the definition of $\pi_{\max}$ (given in (7.17)), $\pi_{\max}(F_{\text{new}}) = c$ implies that $\pi(m, v; F_{\text{new}}) = c$ holds for some covering pair (m, v). By (D.96) and (D.97), $\pi(m, v; F_{\text{new}}) = c$ may be established for at most one pair (m, v). We thus have (D.98).

We now prove (D.99). Consider the execution of line 7. By applying (D.66) with 'j' $\leftarrow m$, 'w' $\leftarrow v$, 'H' $\leftarrow F_{\text{old}}$, '$H'$' $\leftarrow F_{\text{new}}$, and '$r$' $\leftarrow p$, it follows that line 7 establishes $AW^m_v(F_{\text{new}}) = AW^m_v(F_{\text{old}}) - \{r\}$. Since line 6 ensures $r \in AW^m_v(F_{\text{old}})$, we have

$$|AW^m_v(F_{\text{new}})| = |AW^m_v(F_{\text{old}})| - 1,$$

and hence, by the definition of 'req' (given in (7.15)), we have

$$\text{req}(m, v; F_{\text{new}}) = \text{req}(m, v; F_{\text{old}}) - c. \tag{D.102}$$

Moreover, since $AW^m_v(H)$ and $CW^m_v(H)$ are disjoint (by (D.1)), $r \in AW^m_v(F_{\text{old}})$ implies $r \notin CW^m_v(F_{\text{old}})$, and hence, by applying (D.67) as above, we have

$$CW^m_v(F_{\text{new}}) = CW^m_v(F_{\text{old}}). \tag{D.103}$$

Combining (D.102) and (D.103) with the definition of π (given in (7.16)), and using $\pi(m, v; F_{\text{old}}) = c$ (from the antecedent of (D.99)–(D.101)), we have $\pi(m, v; F_{\text{new}}) = 0$, and hence (D.99) follows.

Finally, the antecedent of (D.99)–(D.101) implies that $\pi(j, w; F_{\text{old}}) < c$ holds for each pair $(j, w) \neq (m, v)$. Hence, (D.100) and (D.101) easily follow from (D.96), (D.99), and the definition of $\pi_{\max}$ (given in (7.17)). $\square$

We now prove that (I3) is an invariant. First, consider line 4. Combining (D.96) with the definition of $\pi_{\max}$, it follows that line 4 may increase $\pi_{\max}(F)$ by at most one. Therefore, by (D.97), line 4 establishes $\pi_{\max}(F_{\text{new}}) \leq \pi_{\max}(F_{\text{old}}) + 1 \leq c$, and hence preserves (I3).

Next, consider line 7. By combining (D.98), (D.100), and (D.101), it is easy to see that the loop invariant of the **while** loop (shown before line 6 in Figure D.1) is indeed a correct invariant. Since this loop invariant implies the antecedent of (D.101), it follows that line 7 establishes $\pi_{\max}(F_{\text{new}}) \leq c$, thereby maintaining (I3).

In order to prove that the algorithm constructs the needed computation H', we have yet to show that the algorithm eventually terminates. Note that, by (D.63), each application of Lemma D.6 removes a process from $P(F)$. Since $P(F)$ is initially finite, this procedure cannot continue indefinitely. It follows that the algorithm eventually terminates.

Invariants (I1) and (I3) imply that, after the algorithm terminates, we obtain a regular computation H' in C with induction number m_H, satisfying (D.86). In particular, note that the **while** loop of lines 5–7 establishes $\pi_{\max}(F) \neq c$ upon termination. By (I4), it follows that line 8 establishes $\pi_{\max}(H') < c$, and hence we have (D.86).

Note that every process p_i in K is eventually erased at line 4 (if it has not already been erased via the execution of line 7 with 'r' $\leftarrow p_i$). Thus, by inductively applying (D.63)–(D.67), we have (D.89)–(D.93).

Finally, we claim that H' satisfies (D.87). By (7.17) and (D.84), and since a rank is nonnegative by definition, $\pi(m, v; H) = 0$ holds for every covering pair (m, v). Hence, by (7.18), we have $\pi(H) = 0$. It follows that line 1 establishes $\pi(F) = 0$. We now establish the following two properties.

- The execution of line 4 increases $\pi(F)$ by at most one. (D.104)
- The execution of line 7 decreases $\pi(F)$ by at least $c - 1$. (D.105)

Proof of (D.104)–(D.105)**:** Consider the execution of line s, where s is 4 or 7. By (D.96), one of the following holds.

(i) There exists exactly one pair (j, w) (where $1 \leq j \leq m_H$ and $w \in V$) satisfying $\pi(j, w; F_{\text{new}}) = \pi(j, w; F_{\text{old}}) + 1$. Moreover, for all other pairs $(l, u) \neq (j, w)$ (where $1 \leq l \leq m_H$ and $u \in V$), we have $\pi(l, u; F_{\text{new}}) \leq \pi(l, u; F_{\text{old}})$.

(ii) For all pairs (l, u) (where $1 \leq l \leq m_H$ and $u \in V$), we have $\pi(l, u; F_{\text{new}}) \leq \pi(l, u; F_{\text{old}})$.

We now prove (D.104). If (i) is true, then

$$\begin{aligned}
\pi(F_{\text{new}}) &= \sum_{1\le l\le m_H,\ u\in V} \pi(l,\,u;\,F_{\text{new}}) && \{\text{by (7.18)}\}\\
&= \pi(j,\,w;\,F_{\text{new}}) + \sum_{(l,u)\ne(j,w)} \pi(l,\,u;\,F_{\text{new}})\\
&\le \big(\pi(j,\,w;\,F_{\text{old}})+1\big) + \sum_{(l,u)\ne(j,w)} \pi(l,\,u;\,F_{\text{old}}) && \{\text{by (i)}\}\\
&= \pi(F_{\text{old}})+1, && \{\text{by (7.18)}\}
\end{aligned}$$

and hence we have (D.104). On the other hand, if (ii) is true, then by (7.18), we have $\pi(F_{\text{new}}) \le \pi(F_{\text{old}})$, which in turn implies (D.104).

We now prove (D.105). By (D.99), there exists a covering pair $(m,\,v)$ of F_{old} satisfying $\pi(m,\,v;\,F_{\text{old}}) = c$ and $\pi(m,\,v;\,F_{\text{new}}) = 0$. If (i) is true, then clearly $(j,\,w) \ne (m,\,v)$ holds, and hence

$$\begin{aligned}
\pi(F_{\text{new}}) &= \sum_{1\le l\le m_H,\ u\in V} \pi(l,\,u;\,F_{\text{new}}) && \{\text{by (7.18)}\}\\
&= \pi(j,\,w;\,F_{\text{new}}) + \pi(m,\,v;\,F_{\text{new}}) + \sum_{\substack{(l,u)\ne(j,w)\\(l,u)\ne(m,v)}} \pi(l,\,u;\,F_{\text{new}})\\
&= \pi(j,\,w;\,F_{\text{new}}) + 0 + \sum_{\substack{(l,u)\ne(j,w)\\(l,u)\ne(m,v)}} \pi(l,\,u;\,F_{\text{new}}) && \{\text{by (D.99)}\}\\
&\le \big(\pi(j,\,w;\,F_{\text{old}})+1\big) + 0 + \sum_{\substack{(l,u)\ne(j,w)\\(l,u)\ne(m,v)}} \pi(l,\,u;\,F_{\text{old}}) && \{\text{by (i)}\}\\
&= \big(\pi(j,\,w;\,F_{\text{old}})+1\big) + \big(\pi(m,\,v;\,F_{\text{old}})-c\big) + \sum_{\substack{(l,u)\ne(j,w)\\(l,u)\ne(m,v)}} \pi(l,\,u;\,F_{\text{old}})\\
& && \{\text{by (D.99)}\}\\
&= \pi(F_{\text{old}}) - (c-1), && \{\text{by (7.18)}\}
\end{aligned}$$

and hence we have (D.105). Similarly, if (ii) is true, then we can easily show $\pi(F_{\text{new}}) \le \pi(F_{\text{old}}) - c$, which in turn implies (D.105). □

Since line 4 is executed at most h times, (D.104) implies that $\pi(F)$, being initially zero, increases by at most h $(= |K|)$ throughout the execution of the algorithm. Since

$\pi(F)$ is always nonnegative (by (7.16) and (7.18)), by (D.105), we have the following:

- line 7 may be executed at most $|K|/(c-1)$ times throughout the execution of the algorithm. (D.106)

Moreover, by applying (D.64) with 'j' $\leftarrow m_H$, 'H' $\leftarrow F_{\text{old}}$, and '$H$' $\leftarrow F_{\text{new}}$, it follows that the execution of line 4 may only erase a process in K, and that each execution of line 7 may erase at most one (possibly active) process (*i.e.*, it may reduce $|\text{Act}(F)|$ by at most one). Combining (D.106) with these assertions, (D.87) easily follows. $\square$

In the rest of the proof, we make use of Turán's Theorem [81], stated here again for readers' convenience.

Theorem 5.1 (Turán) *Let $\mathcal{G} = (V, E)$ be an undirected graph with vertex set V and edge set E. If the average degree of $\mathcal{G}$ is d, then an independent set exists with at least $\lceil |V|/(d+1) \rceil$ vertices.* $\square$

Consider a regular computation H. Each process $p \in \text{Act}(H)$ is ready to execute its next critical event $ce(p, m_H + 1)$. As explained in Section 7.3, in order to extend a regular computation, we must eliminate "conflicts" that are caused by appending these critical events. To facilitate this, we define an "conflict-free" computation to be a regular computation in which these conflicts have been eliminated, as follows.

Definition: Consider a regular computation H. For each process $p \in \text{Act}(H)$, define e_p to be its next critical event $ce(p, m_H + 1)$. For each variable $v \in V$, define $Z_v(H)$, the set of active processes that access v in their next critical events, as

$$Z_v(H) = \{p \in \text{Act}(H)\colon e_p \text{ accesses } v\}.$$

Consider two disjoint subsets Z^{Act} and Z^{Cvr} of $\text{Act}(H)$. We say that the pair $(Z^{\text{Act}}, Z^{\text{Cvr}})$ is *conflict-free* in H if (D.107) and one of (C1)–(C3), stated below, are satisfied.

- for each $p \in Z^{\text{Act}} \cup Z^{\text{Cvr}}$, if e_p accesses a variable v, and if H has a single writer q of v, then either $p = q$ or $q \in \text{Act}(H) - (Z^{\text{Act}} \cup Z^{\text{Cvr}})$ holds. (D.107)

Condition (C1): (erasing strategy)

- $Z^{\mathrm{Cvr}} = \{\}$;
- for each $p \in Z^{\mathrm{Act}}$, e_p is an atomic write event, writing a distinct variable;
- for each $p \in Z^{\mathrm{Act}}$, $e_p \neq CS_p$;
- for each $p \in Z^{\mathrm{Act}}$, if e_p writes a variable v, and if a process $q \neq p$ reads v in H, then $q \notin Z^{\mathrm{Act}}$.

Condition (C2): (covering strategy)

- For each variable v, if $Z^{\mathrm{Act}} \cap Z_v(H)$ is nonempty, then $|Z^{\mathrm{Cvr}} \cap Z_v(H)| \geq c \cdot |Z^{\mathrm{Act}} \cap Z_v(H)| + c^2$;
- for each $p \in Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}$, e_p is an atomic write event;
- for each $p \in Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}$, $e_p \neq CS_p$.

Condition (C3): (readers only)

- $Z^{\mathrm{Cvr}} = \{\}$;
- for each $p \in Z^{\mathrm{Act}}$, e_p is a read event. (In particular, $e_p \neq CS_p$.)

We also say that H is *conflict-free* if there exists a partition of $\mathrm{Act}(H)$ into two disjoint sets $Z^{\mathrm{Act}}(H)$ and $Z^{\mathrm{Cvr}}(H)$ (one of which may possibly be empty), such that $\mathrm{Act}(H) = Z^{\mathrm{Act}}(H) \cup Z^{\mathrm{Cvr}}(H)$ and the pair $(Z^{\mathrm{Act}}(H), Z^{\mathrm{Cvr}}(H))$ is conflict-free in H. $\square$

In Lemma D.9, given later, we show that a conflict-free computation H with induction number m_H can be extended to obtain another regular computation G with induction number $m_H + 1$. By extending H, processes in $Z^{\mathrm{Act}}(H)$ become active processes in the extended computation G, and processes in $Z^{\mathrm{Cvr}}(H)$ become new covering processes to cover variables written by processes in $Z^{\mathrm{Act}}(H)$. (Formally, we establish $\mathrm{Act}(G) = Z^{\mathrm{Act}}(H)$ and $\mathrm{Cvr}(G) = \mathrm{Cvr}(H) \cup Z^{\mathrm{Cvr}}(H)$.)

The following lemma shows that, for any regular computation H, we can choose "enough" active processes in H and construct a conflict-free computation F with the same induction number. The strategy chosen to eliminate conflicts determines which condition is established: each of (C1)–(C3) is established by the erasing strategy, the covering strategy, and the "readers only" case, respectively.

Lemma D.8 Let H be a regular computation in C with induction number m_H. Let $n = |\mathrm{Act}(H)|$. Assume the following:

- $m_H \leq c - 2$, (D.108)
- $\pi_{\max}(H) = 0$, and (D.109)

- $n \geq 2.$ (D.110)

Then, there exists a regular computation F in C with induction number m_H, satisfying the following:

- $\pi_{\max}(F) \leq c,$ (D.111)
- F is conflict-free, and (D.112)
- $|Z^{\text{Act}}(F)| \geq \dfrac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)}.$ (D.113)

Proof: For each $p \in \text{Act}(H)$, let e_p be its next critical event $ce(p, m_H+1)$. We claim that $e_p = CS_p$ holds for at most one process p in $\text{Act}(H)$. Assume, to the contrary, that there exist two distinct processes p and q, such that $e_p = CS_p$ and $e_q = CS_q$. Since CS_p does not read any variable, by applying P2′ with 'H' $\leftarrow H\,|\,p$, 'e_p' $\leftarrow CS_p$, and 'G' $\leftarrow H$, we have $H \circ \langle CS_p \rangle \in C$. By applying P2′ again with 'H' $\leftarrow H \mid q$, 'e_p' $\leftarrow CS_q$, and 'G' $\leftarrow H \circ \langle CS_p \rangle$, we have $H \circ \langle CS_p, CS_q \rangle \in C$. However, this contradicts the Exclusion property.

Let $Y_0 = \{p \in \text{Act}(H)\colon e_p \neq CS_p\}$. As shown above, we have $n-1 \leq |Y_0| \leq n$. We partition Y_0 into the sets of "readers" and "writers," and choose the bigger of the two (see Figure 7.10). That is, we choose a set Y such that

$$Y \subseteq \text{Act}(H); \tag{D.114}$$

$$e_p \neq CS_p, \quad \text{for all } p \in Y; \tag{D.115}$$

$$|Y| \geq (n-1)/2; \tag{D.116}$$

$$\big(\forall p : p \in Y :: e_p \text{ is a read event}\big) \quad \vee \quad \big(\forall p : p \in Y :: e_p \text{ is an atomic write event}\big). \tag{D.117}$$

We now erase processes in $\text{Act}(H) - Y$. Since $\text{Act}(H) - Y \subseteq \text{Act}(H)$, by applying Lemma D.4 with 'K' $\leftarrow \text{Act}(H) - Y$, we can construct a regular computation H' in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H') \leq \pi_{\max}(H);$ (D.118)
- for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q,

$$\begin{aligned}
P(H') &= P(H) - (\mathrm{Act}(H) - Y); && \text{(D.119)}\\
\mathrm{Act}^j(H') &= \mathrm{Act}^j(H) - (\mathrm{Act}(H) - Y); && \text{(D.120)}\\
\mathrm{Cvr}^j(H') &= \mathrm{Cvr}^j(H); && \text{(D.121)}\\
AW^j_w(H') &= AW^j_w(H) - (\mathrm{Act}(H) - Y); && \text{(D.122)}\\
CW^j_w(H') &= CW^j_w(H); && \text{(D.123)}\\
RW^j_w(H') &\supseteq RW^j_w(H).
\end{aligned}$$

By (D.109) and (D.118), and since $\pi_{\max}(H')$ is nonnegative by definition, we also have

$$\pi_{\max}(H') = 0. \qquad \text{(D.124)}$$

By (D.114), and by applying (D.120) with 'j' $\leftarrow m_H$, we also have

$$\mathrm{Act}(H') = \mathrm{Act}^{m_H}(H') = \mathrm{Act}^{m_H}(H) - (\mathrm{Act}(H) - Y) = \mathrm{Act}(H) - (\mathrm{Act}(H) - Y) = Y. \qquad \text{(D.125)}$$

We now create a "conflict mapping" $K\colon Y \to P(H') \cup \{\perp\}$, *i.e.*, a mapping indicating which process conflicts with which process, defined over Y. For each $p \in Y$, define $v_{\mathrm{ce}}(p)$ to be the variable accessed by p's next critical event e_p. That is,

- for each $p \in Y$, e_p accesses $v_{\mathrm{ce}}(p)$. (D.126)

For each $p \in Y$, we define $K(p) = q$ if q ($\neq p$) is the single writer of $v_{\mathrm{ce}}(p)$ in H (see Figure 7.9). If H has no single writer of $v_{\mathrm{ce}}(p)$, or if p itself is the single writer of $v_{\mathrm{ce}}(p)$, then we define $K(p) = \perp$. By definition,

$$K(p) \neq p, \quad \text{for all } p \in Y. \qquad \text{(D.127)}$$

We now eliminate conflicts between Y and $\mathrm{Cvr}(H')$ by applying the chain erasing procedure. Define CE, the "covering processes to be erased," as

$$CE = \{K(p)\colon p \in Y \ \wedge \ K(p) \in \mathrm{Cvr}(H')\}.$$

Since $\mathrm{Cvr}(H') \subseteq P(H')$ (by (7.4)), we have

$$|CE| \le |\{K(p)\colon p \in Y\}| \le |Y|, \quad \text{and} \qquad \text{(D.128)}$$

$$CE \subseteq P(H'). \qquad \text{(D.129)}$$

We now apply Lemma D.7 with 'H' $\leftarrow H'$ and 'K' $\leftarrow CE$. Assumptions (D.83)–(D.85) stated in Lemma D.7 follow from (D.108), (D.124), and (D.129), respectively. We thus obtain a regular computation H'' in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H'') < c$; (D.130)
- $|\text{Act}(H'')| \geq |\text{Act}(H') - CE| - |CE|/(c-1)$; (D.131)
- for each segment index j ($1 \leq j \leq m_H$) and variable w,
  - if $AW_w^j(H'')$ is nonempty and $CW_w^j(H'')$ is empty, then $CW_w^j(H')$ is also empty; (D.132)
  - the following hold:

$$P(H'') \subseteq P(H') - CE; \tag{D.133}$$

$$\text{Act}^j(H'') \subseteq \text{Act}^j(H') - CE; \tag{D.134}$$

$$\text{Cvr}^j(H'') \subseteq \text{Cvr}^j(H') - CE; \tag{D.135}$$

$$AW_w^j(H'') \subseteq AW_w^j(H') - CE; \tag{D.136}$$

$$CW_w^j(H'') \subseteq CW_w^j(H') - CE.$$

Since $CE \subseteq \text{Cvr}(H')$ holds by definition, CE and $\text{Act}(H')$ are disjoint by (7.4), and hence, by (D.131), we also have

$$|\text{Act}(H'')| \geq |\text{Act}(H')| - |CE|/(c-1). \tag{D.137}$$

Define

$$Y' = \text{Act}(H''). \tag{D.138}$$

By (D.125), and by applying (D.134) with 'j' $\leftarrow m_H$, we have

$$Y' = \text{Act}^{m_H}(H'') \subseteq \text{Act}^{m_H}(H') - CE = Y - CE. \tag{D.139}$$

Also, by applying (D.137), (D.125), (D.128), and (D.116) (in that order),

$$|Y'| \geq |\text{Act}(H')| - \frac{|CE|}{c-1} = |Y| - \frac{|CE|}{c-1} \geq |Y| - \frac{|Y|}{c-1} = \frac{c-2}{c-1}|Y| \geq \frac{(c-2)(n-1)}{2(c-1)}. \tag{D.140}$$

We now construct an undirected graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y' (see Figure 7.13). For each process p in Y', we introduce edge $\{p, K(p)\}$ if

$K(p) \in Y'$ holds. Since $K(p) \neq p$ (by (D.127)), each edge is properly defined. Since we introduce at most $|Y'|$ edges, the average degree of $\mathcal{G}$ is at most two. Hence, by Theorem 5.1, there exists an independent set $Z \subseteq Y'$ such that

$$|Z| \geq \frac{|Y'|}{3} \geq \frac{(c-2)(n-1)}{6(c-1)}, \tag{D.141}$$

where the latter inequality follows from (D.140). By (D.139), we also have

$$Z \subseteq Y' \subseteq Y - CE. \tag{D.142}$$

Since $Y' = \mathrm{Act}(H'') \subseteq P(H'')$ (by (7.4) and (D.138)), we have $Z \subseteq P(H'')$. Thus, by (D.133), we have

$$Z \subseteq P(H'') \subseteq P(H'). \tag{D.143}$$

We now claim that a process in Z does not conflict with any covering process in H'', or with any other process in Z.

Claim 1: For each $p \in Z$, $K(p) \in P(H'')$ implies $K(p) \in Y' - Z$.

Proof of Claim: Let $q = K(p)$, and assume $q \in P(H'')$. By (7.4), q is either in $\mathrm{Cvr}(H'')$ or $\mathrm{Act}(H'')$. If $q \in \mathrm{Cvr}(H'')$, then by applying (D.135) with 'j' $\leftarrow m_H$, we have $q \in \mathrm{Cvr}(H') - CE$. But, by the definition of CE, we also have $K(p) \notin \mathrm{Cvr}(H') - CE$, a contradiction.

It follows that $q \in \mathrm{Act}(H'')$ holds. Hence, by (D.138) and (D.142), we have $\{p, q\} \subseteq Y'$. Therefore, the edge $\{p, q\}$ $(= \{p, K(p)\})$ is in $\mathcal{G}$ by definition. Since Z is an independent set of $\mathcal{G}$, $p \in Z$ implies $q \notin Z$, and hence the claim follows. $\square$

We now group processes in Z depending on the variables accessed by their next critical events. For each $v \in V$, define Z_v, the set of processes in Z that access v in their next critical events, as

$$Z_v = \{p \in Z\colon v_{\mathrm{ce}}(p) = v\}.$$

Clearly, the sets Z_v form a disjoint partition of Z:

$$\begin{gathered} Z = \bigcup_{v \in V} Z_v, \quad \text{and} \\ Z_v \cap Z_u \neq \{\} \;\Rightarrow\; v = u. \end{gathered} \tag{D.144}$$

Define V_{HC}, the set of variables that experience "high contention" (*i.e.*, those that are accessed by "sufficiently many" next critical events), and V_{LC}, the set of variables that experience "low contention," as

$$\begin{aligned} V_{\text{HC}} &= \{v \in V \colon |Z_v| \geq 4c^2\}, \quad \text{and} \\ V_{\text{LC}} &= \{v \in V \colon 0 < |Z_v| < 4c^2\}. \end{aligned}$$

Then, we have

$$|V_{\text{HC}}| \leq \frac{|Z|}{4c^2}. \tag{D.145}$$

Define P_{HC} (respectively, P_{LC}), the set of processes whose next critical event accesses a variable in V_{HC} (respectively, V_{LC}), as follows:

$$P_{\text{HC}} = \bigcup_{v \in V_{\text{HC}}} Z_v, \quad P_{\text{LC}} = \bigcup_{v \in V_{\text{LC}}} Z_v. \tag{D.146}$$

Since the sets Z_v partition Z, P_{HC} and P_{LC} also partition Z:

$$Z = P_{\text{HC}} \cup P_{\text{LC}} \quad \wedge \quad P_{\text{HC}} \cap P_{\text{LC}} = \{\}. \tag{D.147}$$

We now construct subsets Z^{Act} and Z^{Cvr} of Z, such that the pair $(Z^{\text{Act}}, Z^{\text{Cvr}})$ is conflict-free in H''. (Later, by retaining Z^{Act} and Z^{Cvr}, and erasing all other active processes, we construct a conflict-free computation F satisfying $Z^{\text{Act}}(F) = Z^{\text{Act}}$ and $Z^{\text{Cvr}}(F) = Z^{\text{Cvr}}$.)

Claim 2: There exist two disjoint subsets Z^{Act} and Z^{Cvr} of Z, such that $(Z^{\text{Act}}, Z^{\text{Cvr}})$ is conflict-free in H'', satisfying the following inequality:

$$|Z^{\text{Act}}| \geq \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)}. \tag{D.148}$$

Proof of Claim: By Claim 1, any choice of $(Z^{\text{Act}}, Z^{\text{Cvr}})$ from Z satisfies (D.107). Thus, it suffices to show one of (C1)–(C3). By (D.117) and (D.142), we have either of the following.

- For each $p \in Z$, e_p is a read event. **(R)**
- For each $p \in Z$, e_p is an atomic write event. **(W)**

We consider three cases.

Case 1 (readers only): Condition (R) is true.

Let $Z^{\text{Act}} = Z$ and $Z^{\text{Cvr}} = \{\}$. Condition (R) implies Condition (C3). By (D.141), we have

$$|Z^{\text{Act}}| = |Z| \geq \frac{(c-2)(n-1)}{6(c-1)} \geq \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)},$$

which establishes (D.148).

Case 2 (erasing strategy): Condition (W) is true, and $|P_{\text{HC}}| < |Z|/2$ holds.

By (D.141) and (D.147), we have

$$|P_{\text{LC}}| = |Z| - |P_{\text{HC}}| > \frac{|Z|}{2} \geq \frac{(c-2)(n-1)}{12(c-1)}. \tag{D.149}$$

Note that, by (D.146), P_{LC} is partitioned into nonempty sets Z_v, for each $v \in V_{\text{LC}}$. Moreover, by the definition of V_{LC}, each such Z_v contains less than $4c^2$ processes. Therefore, by (D.149),

$$|V_{\text{LC}}| > \frac{|P_{\text{LC}}|}{4c^2} > \frac{(c-2)(n-1)}{48c^2(c-1)}. \tag{D.150}$$

By the definition of V_{LC}, Z_v is nonempty for each $v \in V_{\text{LC}}$. Thus, we can construct a subset X of P_{LC} that contains exactly one process from each Z_v (for each $v \in V_{\text{LC}}$). Then, by (D.142), (D.147), (D.150), and the definition of Z_v, we have the following:

- $X \subseteq P_{\text{LC}} \subseteq Y'$ $(= \text{Act}(H''))$, (D.151)
- $|X| = |V_{\text{LC}}| > \dfrac{(c-2)(n-1)}{48c^2(c-1)}$, and (D.152)
- $v_{\text{ce}}(p)$ is distinct for each process $p \in X$. (D.153)

As explained in Section 7.3, we want to find a subset of X in which every process p becomes the single writer of $v_{\text{ce}}(p)$. Toward this goal, we now construct an undirected graph $\mathcal{H} = (X, E_{\mathcal{H}})$, where each vertex is a process in X (see Figure 7.14). For each pair $\{p, q\}$ of different processes in X, we introduce edge $\{p, q\}$ if p reads $v_{\mathrm{ce}}(q)$ in H''. By (D.151), each $p \in X$ executes m_H critical events in H'', and hence reads at most m_H distinct variables in H''. Therefore, by (D.153), we collectively introduce at most $|X| \cdot m_H$ edges. Hence, the average degree of $\mathcal{H}$ is at most $2m_H$. Therefore, by Theorem 5.1, there exists an independent set $X' \subseteq X$ such that

$$|X'| \geq \frac{|X|}{2m_H + 1} > \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)}, \tag{D.154}$$

where the latter inequality follows from (D.152). Also, by (D.153),

- $v_{\mathrm{ce}}(p)$ is distinct for each process $p \in X'$, (D.155)

and by (D.126) and the definition of $\mathcal{H}$,

- for each $p \in X'$, if e_p writes a variable v, and if a process $q \neq p$ reads v in H'', then $q \notin X'$. (D.156)

Also, since by (D.115), (D.139), and (D.151), we have the following:

$$e_p \neq CS_p, \quad \text{for each } p \in X'. \tag{D.157}$$

We now define $Z^{\mathrm{Act}} = X'$ and $Z^{\mathrm{Cvr}} = \{\}$. By (D.154), we have (D.148). By (D.155)–(D.157) and Condition (W), we have Condition (C1). Thus, Claim 2 follows.

Case 3 (covering strategy): Condition (W) is true, and $|P_{\mathrm{HC}}| \geq |Z|/2$ holds.

By (D.141), we have

$$|P_{\mathrm{HC}}| \geq \frac{|Z|}{2} \geq \frac{(c-2)(n-1)}{12(c-1)}. \tag{D.158}$$

By (D.146), P_{HC} is partitioned into subsets Z_v, for each $v \in V_{\mathrm{HC}}$. Moreover, by the definition of V_{HC},

$$|Z_v| \geq 4c^2, \quad \text{for each } v \in V_{\mathrm{HC}}. \tag{D.159}$$

Thus, we can partition each such Z_v into two disjoint sets Z_v^{Act} and Z_v^{Cvr}, such that the following holds:

$$Z_v = Z_v^{\text{Act}} \cup Z_v^{\text{Cvr}} \quad \wedge \quad Z_v^{\text{Act}} \cap Z_v^{\text{Cvr}} = \{\}; \tag{D.160}$$

$$|Z_v^{\text{Act}}| = \left\lfloor \frac{|Z_v|}{c^2} \right\rfloor - 1. \tag{D.161}$$

By (D.144) and (D.159)–(D.161), we have the following:

$$Z_v^{\text{Act}} \cap Z_u^{\text{Act}} \neq \{\} \;\Rightarrow\; v = u, \quad \text{for each variable } v \text{ and } u; \tag{D.162}$$

$$|Z_v^{\text{Act}}| > \frac{|Z_v|}{c^2} - 2 \geq \frac{|Z_v|}{c^2} - \frac{|Z_v|}{2c^2} = \frac{|Z_v|}{2c^2}; \tag{D.163}$$

$$|Z_v^{\text{Cvr}}| = |Z_v| - |Z_v^{\text{Act}}| \geq c^2 \cdot (|Z_v^{\text{Act}}| + 1) - |Z_v^{\text{Act}}| = (c^2 - 1) \cdot |Z_v^{\text{Act}}| + c^2. \tag{D.164}$$

We can now define Z^{Act} and Z^{Cvr} as follows:

$$Z^{\text{Act}} = \bigcup_{v \in V_{\text{HC}}} Z_v^{\text{Act}} \quad \text{and} \quad Z^{\text{Cvr}} = \bigcup_{v \in V_{\text{HC}}} Z_v^{\text{Cvr}}. \tag{D.165}$$

By definition, the sets Z^{Act} and Z^{Cvr} partition P_{HC}:

$$P_{\text{HC}} = Z^{\text{Act}} \cup Z^{\text{Cvr}} \quad \wedge \quad Z^{\text{Act}} \cap Z^{\text{Cvr}} = \{\}; \tag{D.166}$$

Also, we have the following:

$$\begin{aligned}
|Z^{\text{Act}}| &= \left|\textstyle\bigcup_{v \in V_{\text{HC}}} Z_v^{\text{Act}}\right| = \textstyle\sum_{v \in V_{\text{HC}}} |Z_v^{\text{Act}}| && \{\text{by (D.162)}\} \\
&> \sum_{v \in V_{\text{HC}}} \frac{|Z_v|}{2c^2} && \{\text{by (D.163)}\} \\
&= \frac{1}{2c^2} \left|\textstyle\bigcup_{v \in V_{\text{HC}}} Z_v\right| \\
& \qquad \{\text{since each } Z_v \text{ is disjoint with each other, by (D.144)}\} \\
&= \frac{1}{2c^2} |P_{\text{HC}}| && \{\text{by (D.146)}\} \\
&\geq \frac{(c-2)(n-1)}{24c^2(c-1)} && \{\text{by (D.158)}\} \\
&> \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)},
\end{aligned}$$

and hence we have (D.148). Finally, by (D.144) and (D.160), the sets Z_v are mutually disjoint, each partitioned into Z_v^{Act} and Z_v^{Cvr}. Thus, by (D.165), it follows that Z^{Act} (respectively, Z^{Cvr}) is a *disjoint* union of Z_v^{Act} (respectively Z_v^{Cvr}) over variables $v \in V_{\mathrm{HC}}$. Hence, we have $Z_v^{\mathrm{Act}} = Z^{\mathrm{Act}} \cap Z_v$ and $Z_v^{\mathrm{Cvr}} = Z^{\mathrm{Cvr}} \cap Z_v$. Thus, by (D.164), we have

$$|Z^{\mathrm{Cvr}} \cap Z_v| \geq (c^2 - 1) \cdot |Z^{\mathrm{Act}} \cap Z_v| + c^2 > c \cdot |Z^{\mathrm{Act}} \cap Z_v| + c^2, \quad \text{(D.167)}$$

where the last inequality follows from $c = \Theta(\log N)$ (given in (7.14)), assuming $N = \omega(1)$.

Also, since $P_{\mathrm{HC}} \subseteq Y$ (by (D.142) and (D.147)), by (D.115), we have the following:

$$e_p \neq CS_p, \quad \text{for each } p \in Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}. \quad \text{(D.168)}$$

Combining (D.167) and (D.168) with Condition (W), we have Condition (C2), and hence Claim 2 follows. □

Define $Z' = Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}$. By (D.142) and Claim 2, we have

$$Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}} = Z' \subseteq Z \subseteq Y' \subseteq Y - CE. \quad \text{(D.169)}$$

We now erase processes in $Y' - Z'$ by applying Lemma D.4 with 'H' $\leftarrow H''$ and 'K' $\leftarrow Y' - Z'$. (Note that $Y' - Z' \subseteq \mathrm{Act}(H'')$ holds since Y' is defined to be $\mathrm{Act}(H'')$.) We thus construct a regular computation F in C with induction number m_H, satisfying assertions (D.170)–(D.175), given below:

- $\pi_{\max}(F) \leq \pi_{\max}(H'')$; (D.170)
- for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q,

$$\begin{aligned} P(F) &= P(H'') - (Y' - Z'); && \text{(D.171)} \\ \mathrm{Act}^j(F) &= \mathrm{Act}^j(H'') - (Y' - Z'); && \text{(D.172)} \\ \mathrm{Cvr}^j(F) &= \mathrm{Cvr}^j(H''); && \text{(D.173)} \\ AW_w^j(F) &= AW_w^j(H'') - (Y' - Z'); && \text{(D.174)} \\ CW_w^j(F) &= CW_w^j(H''). && \text{(D.175)} \end{aligned}$$

By (D.138), (D.169), and by applying (D.172) with 'j' $\leftarrow m_H$, we also have

$$\text{Act}(F) = \text{Act}^{m_H}(F) = \text{Act}^{m_H}(H'') - (Y' - Z') = Y' - (Y' - Z') = Z'. \qquad \text{(D.176)}$$

We now claim that F satisfies (D.111)–(D.113). By (D.130) and (D.170), we have (D.111). Since $(Z^{\text{Act}}, Z^{\text{Cvr}})$ is conflict-free in H'' (by Claim 2), it is also conflict-free in F. Define $Z^{\text{Act}}(F) = Z^{\text{Act}}$ and $Z^{\text{Cvr}}(F) = Z^{\text{Cvr}}$. Since $Z' = Z^{\text{Act}} \cup Z^{\text{Cvr}}$ by definition, by (D.176), $\text{Act}(F)$ is partitioned into two disjoint sets $Z^{\text{Act}}(F)$ and $Z^{\text{Cvr}}(F)$, such that $(Z^{\text{Act}}(F), Z^{\text{Cvr}}(F))$ is conflict-free in F. Thus, F is conflict-free by definition, so we have (D.112). Finally, by (D.148), we have (D.113). □

The following lemma extends a conflict-free computation, thus providing the induction step that leads to the lower bound in Theorem 7.2.

Lemma D.9 Let H be a regular computation in C with induction number m_H. Assume the following:

- $m_H \leq c - 2$, (D.177)
- H is conflict-free, and (D.178)
- $\pi_{\max}(H) \leq c$. (D.179)

Since H is conflict-free, by definition, $\text{Act}(H)$ is partitioned into two disjoint sets $Z^{\text{Act}} = Z^{\text{Act}}(H)$ and $Z^{\text{Cvr}} = Z^{\text{Cvr}}(H)$.

Then, there exists a regular computation $G = H \circ E$ in C with induction number $m_H + 1$, where $E = G^{m_H+1}$ is the newly appended $(m_H + 1)^{st}$ segment, satisfying the following:

- $\pi_{\max}(G) = 0$, and (D.180)
- $\text{Act}(G) = Z^{\text{Act}}$. (D.181)

Proof: As stated above, $\text{Act}(H)$ can be partitioned into two disjoint sets $Z^{\text{Act}} = Z^{\text{Act}}(H)$ and $Z^{\text{Cvr}} = Z^{\text{Cvr}}(H)$, such that $(Z^{\text{Act}}, Z^{\text{Cvr}})$ is a conflict-free pair in H:

$$Z^{\text{Act}} \cup Z^{\text{Cvr}} = \text{Act}(H) \quad \text{and} \quad Z^{\text{Act}} \cap Z^{\text{Cvr}} = \{\}. \qquad \text{(D.182)}$$

For each $p \in \text{Act}(H)$, define e_p, p's next critical event, to be $ce(p, m_H + 1)$. Also define $v_{\text{ce}}(p)$ to be the variable accessed by e_p:

- for each $p \in \text{Act}(H)$, e_p accesses $v_{\text{ce}}(p)$. (D.183)

Define Z_v, the set of active processes that access v via their next critical events, as $Z_v = \{p \in Z\colon v_{\mathrm{ce}}(p) = v\}$. Arbitrarily index processes in Z^{Act} as

$$Z^{\mathrm{Act}} = \{p_1,\ p_2,\ \ldots,\ p_b\}, \tag{D.184}$$

where $b = |Z^{\mathrm{Act}}|$. In order to construct the new $(m_H{+}1)^{\mathrm{st}}$ segment E, for each $p \in Z^{\mathrm{Act}}$, we have to construct its "next covering segment" $C(p, m_H + 1)$, which we denote by $C(p)$. We do this by adding the following events to $C(p)$, for each $p \in Z^{\mathrm{Act}}$ (see Figure 7.6).

Step 1. For each m $(1 \le m \le m_H)$, if $\mathit{ce}(p, m)$ writes a variable v (*i.e.*, $p \in AW_v^m(H)$), and if $CW_v^m(H)$ is nonempty, then we choose a process $\mathit{dp}(p, m)$ from RW_v^m, and deploy $\mathit{dp}(p, m)$ by adding $\mathit{ie}(\mathit{dp}(p, m), m)$, its invocation event on v, to $C(p)$. (After deployment, $\mathit{dp}(p, m)$ does not belong to RW_v^m any more.)

Step 2. If Condition (C2) (from the definition of a conflict-free pair) is true, then let $v = v_{\mathrm{ce}}(p)$. We choose a process $\mathit{dp}(p, m_H{+}1)$ from Z^{Cvr} such that its next critical event, $e_{\mathit{dp}(p,m_H+1)}$, is a write to v. (Thus, we have $p \in Z^{\mathrm{Act}} \cap Z_v$ and $\mathit{dp}(p, m_H{+}1) \in Z^{\mathrm{Cvr}} \cap Z_v$.) We then deploy $\mathit{dp}(p, m_H + 1)$ by adding $\mathit{ie}(\mathit{dp}(p, m_H + 1), m_H + 1)$, its invocation event on v, to $C(p)$.

We claim that there exist enough reserve processes to deploy throughout the construction of all next covering segments. By Lemma D.5, for each covering pair (m, v) of H, we have $|RW_v^m(H)| > |AW_v^m(H)|$. (Assumptions (D.54) and (D.55) stated in Lemma D.5 follow from (D.177) and (D.179), respectively.) Thus, H has enough reserve processes to use in Step 1. Also, if Condition (C2) is true, then we have $|Z^{\mathrm{Cvr}} \cap Z_v| \ge c \cdot |Z^{\mathrm{Act}} \cap Z_v| + c^2 > |Z^{\mathrm{Act}} \cap Z_v|$, and hence we have enough processes in $Z^{\mathrm{Cvr}} \cap Z_v$ to use in Step 2.

We now construct E as follows:

$$E = S(p_1, m_H{+}1) \circ C(p_1) \circ S(p_2, m_H{+}1) \circ C(p_2) \circ \cdots \circ S(p_b, m_H{+}1) \circ C(p_b). \tag{D.185}$$

In order to show that G is a valid computation in C, we first need the following claim. (Informally, we show that G satisfies Lemma D.1.)

Claim 1: Consider an event f_p in G, and a variable v. Denote G as $F_1 \circ \langle f_p \rangle \circ F_2$, where F_1 and F_2 are subcomputations of G. If f_p reads v, then the following holds:

$$\mathit{writer}(v, F_1) = p \ \vee \ \mathit{writer}(v, F_1) = \bot \ \vee \ \mathit{value}(v, F_1) = \star.$$

Proof of Claim: If f_p is an event in H, then by applying Lemma D.1 with 'H' $\leftarrow H$ and 'e_p' $\leftarrow f_p$, the claim follows. Hence, assume that f_p is an event in E.

Since covering segments consist entirely of invocation events, by the definition of E (given in (D.185)),

- f_p is contained in $S(p, m_H + 1)$, (D.186)

and also $p \in Z^{\mathrm{Act}}$ holds. By (D.182), we also have

$$p \in Z^{\mathrm{Act}} \subseteq \mathrm{Act}(H). \tag{D.187}$$

Since f_p reads v, p executes a critical read of v in G. Thus,

- $\mathit{ce}(p, m)$ reads v, for some m $(1 \leq m \leq m_H + 1)$. (D.188)

Let $g_q = \mathit{writer\_event}(v, F_1)$. If we have either $q = p$ or $q = \bot$, then we are done. Thus, assume $q \neq p \ \wedge \ q \neq \bot$. By the Atomicity property, g_q is either an atomic write event of v or an invocation event on v. If g_q is an invocation event, then we have $\mathit{value}(v, F_1) = \star$, and hence we are done.

We claim that g_q cannot be an atomic write event. For the sake of contradiction, assume otherwise, *i.e.*, g_q is an atomic write event of v. Then, by (D.7) and (D.185),

- g_q is contained in solo segment $S(q, l)$, for some l $(1 \leq l \leq m_H{+}1)$. (D.189)

Thus,

- $\mathit{ce}(q, j)$ is a write to v, for some $j \leq l$; (D.190)

We consider two cases, depending on the value of j.

Case 1: $j = m_H + 1$.

In this case, by (D.190),

- $e_q = ce(q, m_H + 1)$ is a write to v. (D.191)

Moreover, by (D.189) and (D.190), g_q is contained in $S(q, m_H + 1)$, and hence, by (D.184) and (D.185), we have

$$q \in Z^{\text{Act}}. \tag{D.192}$$

Thus, by (D.178), (D.191), and the definition of a conflict-free computation,

- either Condition (C1) or Condition (C2) is true,

and hence, by (D.187), $e_p = ce(p, m_H + 1)$ is also a write event. Therefore, by (D.188), we have $m \le m_H$, and hence,

- p reads v in H. (D.193)

By (D.187), (D.191), (D.192), and (D.193), we have a contradiction of (the last line of) Condition (C1). Thus, assume that Condition (C2) is true. By (D.183) and (D.191), it follows that Step 2 (in the construction of E) adds an invocation event h on v to $C(q)$. Therefore, by (D.186), and since g_q precedes f_p, E can be written as follows:

$$E = \cdots \circ S(q, m_H + 1) \circ C(q) \circ \cdots \circ S(p, m_H + 1) \circ \cdots ,$$

where events g_q, h, and f_p are contained in $S(q, m_H + 1)$, $C(q)$, and $S(p, m_H + 1)$, respectively. Therefore, G contains a write on v (namely, h) *between* g_q and f_p, which contradicts $g_q = \mathit{writer\_event}(v, F_1)$.

Case 2: $1 \le j \le m_H$.

In this case, by (D.190), q writes v in H, and $q \in AW_v^j(H)$ holds. We consider two cases.

First, assume that $CW_v^j(H)$ is empty, *i.e.*,

- q is the single writer of v in H. (D.194)

If $1 \le m \le m_H$, then by (D.188), and by applying R4 with 'p' $\leftarrow q$, 'q' $\leftarrow p$, and 'm' $\leftarrow j$, we have $p \in \text{Cvr}^j(H)$. However, since $\text{Cvr}^j(H) \subseteq \text{Cvr}(H)$ (by

(7.8)), and since Act(H) and Cvr(H) are disjoint (by (7.4)), this contradicts (D.187).

Therefore, by (D.188), $m = m_H + 1$ holds, and $e_p = ce(p, m_H + 1)$ reads v. Thus, by (D.107) and (D.194), we have $q \in \mathrm{Act}(H) - (Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}})$. However, this is impossible by (D.182).

Second, assume that q is not the single writer of H, *i.e.*, $CW_v^j(H)$ is nonempty. If $l \leq m_H$, then by (D.189), and by applying R2 with 'm' $\leftarrow l$, it follows that $C(q, l)$ contains an invocation event h on v. On the other hand, if $l = m_H + 1$, then by (D.184), (D.185), and (D.189), we have $q \in Z^{\mathrm{Act}}$, and hence, Step 1 adds an invocation event h on v to $C(q) = C(q, m_H + 1)$. Therefore, by (D.186), and since g_q precedes f_p, G can be written as follows:

$$G = \cdots \circ S(q, l) \circ C(q, l) \circ \cdots \circ S(p, m_H + 1) \circ \cdots ,$$

where events g_q, h, and f_p are contained in $S(q, l)$, $C(q, l)$, and $S(p, m_H + 1)$, respectively. Therefore, G contains a write on v (namely, h) *between* g_q and f_p, which contradicts $g_q = \mathit{writer\_event}(v, F_1)$. □

Claim 1 implies that each process $p \in Z^{\mathrm{Act}}$ cannot distinguish its execution in $G = H \circ E$ from its solo computation. Thus, each such p can execute its next solo segment after H. Moreover, all events in the next covering segments are invocation events, and hence they cannot read any variable. Thus, by inductively applying P2$'$, we can easily show that G is a valid computation in C.

We now claim that G is a regular computation with induction number $m_H + 1$, satisfying the following for each segment index m $(1 \leq m \leq m_H + 1)$ and variable v:

$$\mathrm{Act}^m(G) = \begin{cases} \mathrm{Act}^m(H), & \text{if } m \leq m_H \\ Z^{\mathrm{Act}}, & \text{if } m = m_H + 1; \end{cases} \tag{D.195}$$

$$\mathrm{Cvr}^m(G) = \begin{cases} \mathrm{Cvr}^m(H), & \text{if } m \leq m_H \\ \mathrm{Cvr}(H) \cup Z^{\mathrm{Cvr}}, & \text{if } m = m_H + 1; \end{cases} \tag{D.196}$$

$$AW_v^m(G) = \begin{cases} AW_v^m(H), & \text{if } m \leq m_H \\ \{\}, & \text{if } m = m_H + 1 \text{ and Condition (C3) is true} \\ Z^{\mathrm{Act}} \cap Z_v, & \text{if } m = m_H + 1 \text{ and Condition (C1) or (C2) is true;} \end{cases} \tag{D.197}$$

$$CW_v^m(G) \;=\; \begin{cases} CW_v^m(H), & \text{if } m \le m_H \\ Z^{\mathrm{Cvr}} \cap Z_v, & \text{if } m = m_H + 1 \text{ and Condition (C2) is true} \\ \{\}, & \text{if } m = m_H + 1 \text{ and Condition (C1) or (C3) is true.} \end{cases} \tag{D.198}$$

From (D.185) and the construction of the next covering segments, assertions (7.3)–(7.13) and (D.195)–(D.198) follow immediately. The construction of the next covering segments ensures that G satisfies R1 and R2. From (D.178) and the definition of a conflict-free computation, it follows that each next critical event e_p (that is in E) is different from CS_p, and hence we have R3.

We now claim that G satisfies R4. Consider some segment index m $(1 \le m \le m_H + 1)$, process p, and variable v, such that $p \in AW_v^m(G)$ and $CW_v^m(G)$ is empty. We consider two cases.

First, assume $m \le m_H$. Note that, in this case, we have $AW_v^m(G) = AW_v^m(H)$ and $CW_v^m(G) = CW_v^m(H)$. (Thus, p is the single writer of v in H by definition.) By applying R4 to H, it follows that, for each segment index j $(1 \le j \le m_H)$ and each process $q \in \mathrm{Act}^j(G) = \mathrm{Act}^j(H)$ different from p, the following hold:

(i) if $j < m$ and $ce(q, j)$ is a write to v, then $CW_v^j(H)$ is nonempty;
(ii) if $j < m$ and $ce(q, j)$ is a read of v, then $q \in \mathrm{Cvr}^m(H)$ holds;
(iii) if $m \le j \le m_H$, then $ce(q, j)$ does not access v.

Thus, in order to prove that m, p, and v satisfy R4, it suffices to assume $j = m_H + 1$ and consider a process $q \in \mathrm{Act}^{m_H+1}(G)$. Our proof obligation is to show that $ce(q, m_H + 1)$ does not access v.

For the sake of contradiction, assume otherwise. By (D.178), $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ satisfies (D.107) with respect to H. By (D.195), we have $q \in Z^{\mathrm{Act}}$. Thus, applying (D.107) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, and using $p \ne q$ and the fact that p is the single writer of v in H, we have $q \in \mathrm{Act}(H) - (Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}})$, but this is impossible by (D.182).

Second, assume that $m = m_H + 1$. Since $p \in \mathrm{Act}^{m_H+1}(G)$, by (D.195), we have $p \in Z^{\mathrm{Act}}$. Since $e_p = ce(p, m_H + 1)$ writes v and $CW_v^m(G)$ is empty, Condition (C1) must be true. Consider a segment index j $(1 \le j \le m_H + 1)$ and a process $q \in \mathrm{Act}^j(G)$ different from p. Our proof obligation is to show the following three conditions:

(i) if $j \le m_H$ and $ce(q, j)$ is a write to v, then $CW_v^j(G)$ is nonempty;
(ii) if $j \le m_H$ and $ce(q, j)$ is a read of v, then $q \in \mathrm{Cvr}^{m_H+1}(G)$ holds;
(iii) if $j = m_H + 1$, then $ce(q, j)$ does not access v.

Proof of (i)–(iii): First, consider (i). For the sake of contradiction, assume that $CW_v^j(G)$ is empty. Thus, q is a single writer of v in H. As shown above, $(Z^{\text{Act}}, Z^{\text{Cvr}})$ satisfies (D.107) with respect to H, and hence we have either $p = q$ or $q \in \text{Act}(H) - (Z^{\text{Act}} \cup Z^{\text{Cvr}})$. The former contradicts our assumption, and the latter is impossible by (D.182).

Second, consider (ii). In this case, the last line of Condition (C1) implies $q \notin Z^{\text{Act}}$, which in turn implies $q \in \text{Cvr}^{m_H+1}(G)$ by (D.182), (D.195), and (D.196).

Finally, consider (iii). By (D.195), $q \in \text{Act}^{m_H+1}(G)$ implies $q \in Z^{\text{Act}}$. Combining this with $p \in Z^{\text{Act}}$, and using the second line of Condition (C1), (iii) follows easily.

Finally, we claim that G satisfies (D.180) and (D.181). From (D.195), we have (D.181). (Note that Z^{Act} is defined to be $Z^{\text{Act}}(H)$.) In order to show (D.180), we must show $\pi(m, v; G) = 0$ for every covering pair (m, v) in G. We consider two cases.

First, if $m \le m_H$, then by (D.179), we have

$$\pi(m, v; H) \le c.$$

Since $m \le m_H$, we have $AW_v^m(G) = AW_v^m(H)$ and $CW_v^m(G) = CW_v^m(H)$. Hence, by (7.15), and since G has an induction number of $m_H + 1$, we also have

$$\text{req}(m, v; G) = \text{req}(m, v; H) - c.$$

Combining these two assertions with the definition of 'π' (given in (7.16)), we have $\pi(m, v; G) = 0$.

Second, if $m = m_H + 1$, then (m, v) is a covering pair only if Z^{Cvr} is nonempty, *i.e.*, only if Condition (C2) is true. Moreover, by (D.197) and (D.198), we have $AW_v^m(G) = Z^{\text{Act}} \cap Z_v$ and $CW_v^m(G) = Z^{\text{Cvr}} \cap Z_v$. Hence, by (C2), we have $|CW_v^m(G)| \ge c \cdot |AW_v^m(G)| + c^2 > c \cdot (|AW_v^m(G)| + c - m_H) = \text{req}(m, v; G)$, and hence $\pi(m, v; G) = 0$ follows. $\square$

Theorem 7.2 *For any one-shot mutual exclusion system $\mathcal{S} = (C,\ P,\ V)$, there exist a p-computation F such that F does not contain CS_p, and p executes $\Omega(\log N/\log\log N)$ critical events in F, where $N = |P|$.*

Proof: Let $H_1 = S(1,1) \circ S(2,1) \circ \cdots \circ S(N,1) = \langle \mathit{Enter}_1,\ \mathit{Enter}_2,\ \ldots,\ \mathit{Enter}_N \rangle$, where $P = \{1, 2, \ldots, N\}$. By the definition of a mutual exclusion system, $H_1 \in C$. In H_1, each process p becomes a "single writer" of its auxiliary variable entry_p. By checking conditions (7.3)–(7.13) and R1–R4 individually, it follows that H_1 is a regular computation with induction number 1.

We repeatedly apply Lemma D.8 and Lemma D.9, and we can construct a sequence of computations $H_1,\ H_2,\ \ldots,\ H_k$, such that each computation H_m has induction number m. We stop the induction at step k when assumption (D.108) or (D.110) of Lemma D.8 is not satisfied.

Define $n_m = |\mathrm{Act}(H_m)|$ for each m. Applying Lemma D.8 with 'H' $\leftarrow H_m$, we construct a conflict-free computation F_m satisfying

$$|Z^{\mathrm{Act}}(F_m)| \geq \frac{(c-2)(n_m-1)}{48c^2(c-1)(2m+1)}.$$

(This inequality follows from (D.113).) Applying Lemma D.9 with 'H' $\leftarrow F_m$, we construct H_{m+1} such that $\mathrm{Act}(H_{m+1}) = Z^{\mathrm{Act}}(F_m)$ (by (D.181)). Combining these relations, we have

$$n_{m+1} \geq \frac{(c-2)(n_m-1)}{48c^2(c-1)(2m+1)},$$

and hence, by (7.14), (D.108), and (D.110),

$$n_{m+1} \geq \frac{a' n_m}{m \log^2 N} \geq \frac{a n_m}{\log^3 N},$$

where a and a' are some fixed constants. This in turn implies

$$\log n_{m+1} \geq \log n_m - 3 \log\log N + \log a.$$

Therefore, by iterating over $1 \leq m < k$, and using $n_1 = N$, we have

$$\log n_k \geq \log N - 3(k-1)\log\log N + (k-1)\log a. \tag{D.199}$$

If we stop the induction at step k because assumption (D.108) is not satisfied, then we have $k = c-2$, and hence, by (7.14), $k = \Theta(\log N)$ holds. On ther other hand, if we stop the induction because assumption (D.110) is not satisfied, then we have $n_k \leq 1$, and hence, by (D.199),

$$0 \geq \log N - 3(k-1) \log\log N + (k-1) \log a,$$

which in turn implies

$$k \geq \frac{\log N}{3 \log\log N - \log a} + 1 = \Theta\left(\frac{\log N}{\log\log N}\right).$$

Therefore, in either case, we have $k = \Omega(\log N/ \log\log N)$. Since H_{k-1} satisfies (D.110), we can choose a process p from $\mathrm{Act}(H_{k-1})$. Since p executes exactly $k-1$ solo segments (and hence, $k-1$ critical events) in H_{k-1}, $H_{k-1} \mid p$ is a solo computation that satisfies the theorem. □

APPENDIX E

CORRECTNESS PROOF FOR ALGORITHM G-CC IN SECTION 8.2.1

In this appendix, we formally prove that ALGORITHM G-CC, presented in Section 8.2.1, satisfies the Exclusion and Starvation-freedom properties. Our proof makes use of a number of auxiliary variables. In Figure E.1, ALGORITHM G-CC is shown with these added auxiliary variables, which are accessed only by statements 5 and 18. We begin with a description of these variables.

Private auxiliary variable $p.\mathsf{position}$ keeps track of p's position in the queue; as shown by invariants (I14), (I15), (I26), and (I27) given below, when p is in its exit section, $p.\mathsf{position}$ equals the non-auxiliry variable $p.pos$.

For $i = 0$ or 1, $\mathsf{HistLen}[i]$, $\mathsf{Hist}[i][0..\infty]$, and $\mathsf{Param}[i][0..\infty]$ keep the history of queue i since the last time it was initialized. (As explained in Section 8.2.1, each queue is accessed at most $2N$ times before it is re-initialized. Thus, only finite prefixes of $\mathsf{Hist}[i][\ldots]$ and $\mathsf{Param}[i][\ldots]$ are actually used.)

Example. Assume that queue i is initially empty. Then, initially we have the following.

$$\begin{aligned} Tail[i] &= \bot; \\ \mathsf{HistLen}[i] &= 0; \\ \mathsf{Hist}[i] &= (\bot,\ \bot,\ \bot,\ \ldots); \\ \mathsf{Param}[i] &= (\bot,\ \bot,\ \bot,\ \ldots). \end{aligned}$$

Now suppose that processes p, q, r, and p execute statement 5 (in that order), and that the private variables $p.counter$, $q.counter$, and $r.counter$ initially equal 0, 3, and 5, respectively.

```
shared variables
    $CurrentQueue$: 0..1;
    $Tail$: array[0..1] of $Vartype$ initially $\bot$;
    $Position$: array[0..1] of $0..2N-1$ initially 0;
    $Signal$: array[0..1][$Vartype$] of boolean initially $false$;
    $Active$: array[$0..N-1$] of boolean initially $false$;
    $QueueIdx$: array[$0..N-1$] of $(\bot, 0..1)$

private variables
    $idx$: 0..1;
    $counter$: integer;
    $prev, self, tail$: $Vartype$;
    $pos$: $0..2N-1$

private auxiliary variable
    position: $0..\infty$

type
    ParamType = record
        $proc$: $0..N-1$;
        $counter$: integer
    end

shared auxiliary variables
    HistLen: array[0..1] of $0..\infty$ initially 0;
    Hist: array[0..1][$0..\infty$] of $Vartype$ initially $\bot$;
    Param: array[0..1][$0..\infty$] of ParamType initially $\bot$

process $p$ ::    /* $0 \le p < N$ */
while $true$ do
0:   Noncritical Section;
1:   $QueueIdx[p] := \bot$;
2:   $Active[p] := true$;
3:   $idx := CurrentQueue$;
4:   $QueueIdx[p] := idx$;
5:   /* atomically execute lines 5a–5h */
     5a:  $prev := Tail[idx]$;
     5b:  $self := \phi(prev, \alpha_p[counter])$;
     5c:  $Tail[idx] := self$;
     5d:  position := HistLen[$idx$];
     5e:  Param[$idx$][position] := $(p, counter)$;
     5f:  Hist[$idx$][position + 1] := $self$;
     5g:  HistLen[$idx$] := position + 1;
     5h:  $counter := counter + 1$;
     if $prev \ne \bot$ then
6:       await $Signal[idx][prev]$;
7:       $Signal[idx][prev] := false$
     fi;
8:   Entry$_2$($idx$);
9:   Critical Section;
10:  $pos := Position[idx]$;
11:  $Position[idx] := pos + 1$;
12:  Exit$_2$($idx$);
13:  if $(pos < N) \wedge (pos \ne p)$ then
14:      await $\neg Active[pos] \vee$
15:            $(QueueIdx[pos] = idx)$
     elseif $pos = N$ then
16:      $tail := Tail[1-idx]$;
17:      $Signal[1-idx][tail] := false$;
18:  /* atomically execute lines 18a–18d */
     18a: $Tail[1-idx] := \bot$;
     18b: HistLen[$1-idx$] := 0;
     18c: forall $j$ do Param[$1-idx$][$j$] := $\bot$ od;
     18d: forall $j$ do Hist[$1-idx$][$j$] := $\bot$ od;
19:      $Position[1-idx] := 0$;
20:      $CurrentQueue := 1-idx$
     fi;
21:  $Signal[idx][self] := true$;
22:  $Active[p] := false$
od
```

Figure E.1: Algorithm G-CC with auxiliary variables added.

If the underlying *fetch-and-ϕ* primitive is *fetch-and-store* with $2N+1$ distinct values (as defined on page 213), then we have the following after these four *fetch-and-ϕ* invocations take place.

$$\begin{aligned} \mathit{Tail}[i] &= (p,1); \\ \mathsf{HistLen}[i] &= 4; \\ \mathsf{Hist}[i] &= \big(\bot,\ (p,0),\ (q,1),\ (r,1),\ (p,1),\ \bot,\ \bot,\ \ldots\big); \\ \mathsf{Param}[i] &= \big((p,0),\ (q,3),\ (r,5),\ (p,1),\ \bot,\ \bot,\ \ldots\big). \end{aligned}$$

On the other hand, if the underlying *fetch-and-ϕ* primitive is *fetch-and-increment* (where we define $\bot = 0$), then $\mathit{Tail}[i]$ and $\mathsf{Hist}[i]$ have the following final value, while the other two variables hold the same values as above.

$$\begin{aligned} \mathit{Tail}[i] &= 4; \\ \mathsf{Hist}[i] &= \big(\bot\ (=0),\ 1,\ 2,\ 3,\ 4,\ \bot,\ \bot,\ \ldots\big). \end{aligned}$$

E.1 List of Invariants

We will establish the Exclusion property by proving that the conjunction of a number of assertions is an invariant. This proves that each of these assertions individually is an invariant. These invariants are listed below. Unless stated otherwise, we assume the following: i ranges over 0 and 1; h, j, k, and l range over $0..\infty$; x and y range over *Vartype*; p, q, and r range over $0..N-1$.

invariant (Exclusion) $\big|\{p :: p@\{9..12\}\}\big| \le 1$ (I1)

invariant $\big|\{p :: p@\{7..21\} \wedge p.\mathit{idx} = i\}\big| \le 1$ (I2)

invariant $p@\{6..22\} \wedge p.\mathit{idx} = i \wedge p.\mathsf{position} = h \Rightarrow$
$\mathsf{HistLen}[i] > h \wedge p.\mathit{prev} = \mathsf{Hist}[i][h] \wedge p.\mathit{self} = \mathsf{Hist}[i][h+1]$ (I3)

invariant $0 < \mathsf{HistLen}[i] < 2N \Rightarrow \mathit{Tail}[i] \ne \bot$ (I4)

invariant $p@\{8..21\} \wedge p.\mathit{idx} = i \Rightarrow (\forall x :: \mathit{Signal}[i][x] = \mathit{false})$ (I5)

invariant $\mathit{Signal}[i][x] = \mathit{true} \Rightarrow x = \mathsf{Hist}[i][\mathit{Position}[i]]$ (I6)

invariant $\mathit{Signal}[i][x] = \mathit{Signal}[i][y] = \mathit{true} \Rightarrow x = y$ (I7)

invariant $p@\{7\} \wedge p.\mathit{idx} = i \Rightarrow \mathit{Signal}[i][p.\mathit{prev}] = \mathit{true}$ (I8)

invariant $\big|\{p :: p@\{6,7\} \wedge p.\mathit{idx} = i \wedge p.\mathit{prev} = x\}\big| \le 1$ (I9)

invariant $\mathit{Position}[i] = q + 1 \;\Rightarrow$
$$q@\{0..3\} \;\vee\; q.\mathit{idx} = i \;\vee\; (\exists p :: p@\{12..20\} \;\wedge\; p.\mathit{idx} = i) \qquad \text{(I10)}$$

invariant $q + 1 < \mathit{Position}[i] \le N \;\Rightarrow\; q@\{0..3\} \;\vee\; q.\mathit{idx} = i$ (I11)

invariant $p@\{11..13\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; p.\mathit{pos} = N \;\Rightarrow$
$$\mathit{CurrentQueue} = i \;\wedge\; (\forall q :: q@\{0..3\} \;\vee\; q.\mathit{idx} = i) \qquad \text{(I12)}$$

invariant $p@\{16..20\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow$
$$\mathit{CurrentQueue} = i \;\wedge\; (\forall q :: q@\{0..3\} \;\vee\; q.\mathit{idx} = i) \qquad \text{(I13)}$$

invariant $p@\{11\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; \mathit{Position}[i] = p.\mathit{pos}$ (I14)

invariant $p@\{12..21\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; \mathit{Position}[i] = p.\mathit{pos} + 1$ (I15)

invariant $\mathsf{HistLen}[i] > h \;\wedge\; \mathsf{Param}[i][h] = (p, c) \;\Rightarrow$
$$\mathsf{Hist}[i][h+1] = \phi(\mathsf{Hist}[i][h],\; \alpha_p[c]) \qquad \text{(I16)}$$

invariant $\mathit{Tail}[i] = \mathsf{Hist}[i][\mathsf{HistLen}[i]]$ (I17)

invariant $\mathsf{Hist}[i][0] = \perp$ (I18)

invariant $0 \le j < k < \mathsf{HistLen}[i] \;\wedge$
$\mathsf{Param}[i][j] = (p, c_1) \;\wedge\; \mathsf{Param}[i][k] = (p, c_2) \;\wedge$
$(\forall l : j < l < k :: \mathsf{Param}[i][l].\mathit{proc} \ne p) \;\Rightarrow$
$$c_2 = c_1 + 1 \qquad \text{(I19)}$$

invariant $0 \le j < \mathsf{HistLen}[i] \;\wedge$
$\mathsf{Param}[i][j] = (p, c) \;\wedge$
$(\forall k : j < k < \mathsf{HistLen}[i] :: \mathsf{Param}[i][k].\mathit{proc} \ne p) \;\Rightarrow$
$(p.\mathit{counter} = c + 1 \;\wedge\; p@\{4..22\} \;\wedge\; p.\mathit{idx} = i) \;\vee$
$(p.\mathit{counter} = c + 1 \;\wedge\; p@\{0..3\}) \;\vee$
$(p@\{0..3\} \;\wedge\; \mathit{CurrentQueue} = 1 - i) \;\vee$
$(p@\{4..22\} \;\wedge\; p.\mathit{idx} = \mathit{CurrentQueue} = 1 - i)$ (I20)

invariant $0 \le \mathsf{HistLen}[i] \le 2N$ (I21)

invariant $0 \le \mathit{Position}[i] \le 2N$ (I22)

invariant $\mathsf{HistLen}[i] = 0 \;\Rightarrow\; (\forall x :: \mathit{Signal}[i][x] = \mathit{false})$ (I23)

invariant $p@\{18\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; (\forall x :: \mathit{Signal}[1-i][x] = \mathit{false})$ (I24)

invariant $p@\{17\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; p.\mathit{tail} = \mathit{Tail}[1-i]$ (I25)

invariant $p@\{7..11\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; \mathit{Position}[i] = p.\mathsf{position}$ (I26)

invariant $p@\{12..21\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; \mathit{Position}[i] = p.\mathsf{position} + 1$ (I27)

invariant $\big[\, \mathsf{HistLen}[i] - \mathit{Position}[i] = \big|\{p :: p@\{6..11\} \;\wedge\; p.\mathit{idx} = i\}\big| \,\big] \;\vee$
$(\exists p :: p@\{19\} \;\wedge\; p.\mathit{idx} = 1 - i)$ (I28)

invariant $\big|\{p :: p@\{6..21\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; p.\mathsf{position} = h\}\big| \le 1$ (I29)

invariant $\mathit{CurrentQueue} = i \;\Rightarrow$

$$\big|\{p :: p@\{4,5\} \ \wedge\ p.idx = 1-i\}\big| \le 2N - \mathsf{HistLen}[1-i] \tag{I30}$$

invariant $\mathit{Position}[i] = N+1 \ \Rightarrow$
$$\mathit{CurrentQueue} = 1-i \ \vee$$
$$(\exists p :: p@\{12..20\} \ \wedge\ p.idx = i \ \wedge\ p.pos = N) \tag{I31}$$

invariant $\mathit{Position}[i] > N+1 \ \Rightarrow\ \mathit{CurrentQueue} = 1-i$ (I32)

invariant $\mathit{Active}[p] = p@\{3..22\}$ (I33)

invariant $p@\{5..22\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{QueueIdx}[p] = i$ (I34)

invariant $p@\{2..4\} \ \Rightarrow\ \mathit{QueueIdx}[p] = \bot$ (I35)

invariant $\mathit{Position}[i] \le N \ \Rightarrow$
$$\mathit{CurrentQueue} = i \ \vee$$
$$(\exists p :: p@\{20\} \ \wedge\ p.idx = 1-i) \ \vee$$
$$\big(\mathit{Position}[i] = 0 \ \wedge\ (\forall q :: q@\{0..3\} \ \vee\ q.idx = 1-i)\big) \tag{I36}$$

invariant $1 \le \mathit{Position}[i] \le N \ \Rightarrow\ \mathit{CurrentQueue} = i$ (I37)

invariant $p@\{4,5\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathsf{HistLen}[i] < 2N$ (I38)

invariant $p@\{19,20\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathsf{HistLen}[1-i] = 0$ (I39)

invariant $p@\{20\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Position}[1-i] = 0$ (I40)

invariant $\mathit{Position}[i] \le h < \mathsf{HistLen}[i] \ \Rightarrow$
$$(\exists p :: p@\{6..11\} \ \wedge\ p.idx = i \ \wedge\ p.\mathsf{position} = h) \tag{I41}$$

invariant $\mathit{Position}[i] = h > 0 \ \Rightarrow$
$$\mathit{Signal}[i][\mathsf{Hist}[i][h]] = \mathit{true} \ \vee$$
$$\big(\exists p :: p@\{12..21\} \ \wedge\ p.idx = i \ \wedge\ p.\mathsf{position} = h-1\big) \ \vee$$
$$\big(\exists q :: q@\{8..11\} \ \wedge\ q.idx = i \ \wedge\ q.\mathsf{position} = h\big) \tag{I42}$$

invariant $0 < j < k \le \mathsf{HistLen}[i] \ \wedge\ k < 2N \ \wedge$
$\mathsf{Param}[i][j-1].\mathit{proc} \ne \mathsf{Param}[i][k-1].\mathit{proc} \ \Rightarrow$
$$\mathsf{Hist}[i][j] \ne \mathsf{Hist}[i][k] \tag{I43}$$

invariant $0 < j \le \mathsf{HistLen}[i] \ \wedge\ j < 2N \ \Rightarrow\ \mathsf{Hist}[i][j] \ne \bot$ (I44)

invariant $0 < j < k \le \mathsf{HistLen}[i] \ \wedge\ k < 2N \ \wedge$
$\mathsf{Param}[i][j-1] = (p,c) \ \wedge\ \mathsf{Param}[i][k-1] = (p,c+1) \ \Rightarrow$
$$\mathsf{Hist}[i][j] \ne \mathsf{Hist}[i][k] \tag{I45}$$

invariant $\mathit{Position}[i] \le h < \mathsf{HistLen}[i] \ \wedge\ \mathsf{Param}[i][h].\mathit{proc} = p \ \Rightarrow$
$$p@\{6..11\} \ \wedge\ p.idx = i \ \wedge\ p.\mathsf{position} = h \tag{I46}$$

invariant $p@\{6\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Position}[i] \le p.\mathsf{position}$ (I47)

invariant $\mathit{Position}[i] \le j < k < \mathsf{HistLen}[i] \ \Rightarrow\ \mathsf{Hist}[i][j] \ne \mathsf{Hist}[i][k]$ (I48)

E.2 Proof of the Exclusion Property

We now prove that each of (I1)–(I48) is an invariant. For each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.) If I is an implication (which is the case for most of our invariants), then it suffices to check only those program statements that may establish the antecedent of I, or that may falsify the consequent if executed while the antecedent holds.

invariant (Exclusion) $\big|\{p :: p@\{9..12\}\}\big| \leq 1$ (I1)

Proof: Since the $\texttt{Entry}_2$ and $\texttt{Exit}_2$ routines (statements 8 and 12) are assumed to be correct, (I1) follows easily from (I2). □

invariant $\big|\{p :: p@\{7..21\} \ \wedge \ p.idx = i\}\big| \leq 1$ (I2)

Proof: The only statements that may potentially falsify (I2) are 5.p and 6.p. Statement 5.p may falsify (I2) by establishing $p@\{7..21\} \ \wedge \ p.idx = i$ only if executed when

$$p.idx = i \ \wedge \ Tail[i] = \bot \ \wedge \ q@\{7..21\} \ \wedge \ q.idx = i$$

holds, where q is any arbitrary process, different from p. (In this case, process p transits from statement 5 to statement 8.) By (I4) and (I38), $p@\{5\} \wedge p.idx = i \wedge Tail[i] = \bot$ implies $\mathsf{HistLen}[i] = 0$. However, by applying (I3) with 'p' $\leftarrow q$, we have $\mathsf{HistLen}[i] > q.\mathsf{position} \geq 0$, a contradiction. Thus, statement 5.p cannot falsify (I2).

Statement 6.p may falsify (I2) by establishing $p@\{7..21\} \wedge p.idx = i$ only if executed when

$$p.idx = i \ \wedge \ Signal[i][p.prev] = true \ \wedge \ q@\{7..21\} \ \wedge \ q.idx = i,$$

holds, where q is any arbitrary process, different from p. By (I5), we have $q@\{7\}$. Thus, by (I8), we have $Signal[i][q.prev] = true$, which in turn implies $p.prev = q.prev$ by (I7). However, by (I9), this implies $p = q$, a contradiction. Thus, statement 6.p cannot falsify (I2). □

invariant $p@\{6..22\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h \ \Rightarrow$

$$\mathsf{HistLen}[i] > h \ \wedge \ p.prev = \mathsf{Hist}[i][h] \ \wedge \ p.self = \mathsf{Hist}[i][h+1] \qquad \text{(I3)}$$

Proof: The only statement that may establish the antecedent is $5.p$, which may do so only if executed when $p.idx = i \wedge \mathsf{HistLen}[i] = h$ holds. In this case, $5.p$ establishes $\mathsf{HistLen}[i] = h+1 \wedge p.self = \mathsf{Hist}[i][h+1]$. Moreover, by (I17), $Tail[i] = \mathsf{Hist}[i][h]$ holds before $5.p$ is executed, and hence, $p.prev = \mathsf{Hist}[i][h]$ holds afterward.

The only statements that may falsify the consequent are $5.q$ and $18.q$, where q is any arbitrary process. As shown above, statement $5.p$ cannot falsify (I3). For $q \neq p$, if statement $5.q$ is executed when $\mathsf{HistLen}[i] > h$ holds, then it preserves $\mathsf{HistLen}[i] > h$, and does not update either $\mathsf{Hist}[i][h]$ or $\mathsf{Hist}[i][h+1]$. Thus, statement $5.q$ preserves (I3).

Statement $18.q$ may falsify the consequent only if executed when $q.idx = 1-i$ holds. However, by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1-i$, this implies $p@\{0..3\} \vee p.idx = 1-i$. Thus, in this case, the antecedent is false before and after the execution of $18.q$. $\square$

invariant $0 < \mathsf{HistLen}[i] < 2N \Rightarrow Tail[i] \neq \bot$ (I4)

Proof: Invariant (I4) follows easily by applying (I44) with 'j' $\leftarrow \mathsf{HistLen}[i]$, and using (I17). $\square$

invariant $p@\{8..21\} \wedge p.idx = i \Rightarrow (\forall x :: Signal[i][x] = false)$ (I5)

Proof: The only statements that may establish the antecedent are $5.p$ and $7.p$. Statement $5.p$ may establish the antecedent only if executed when $p.idx = i \wedge Tail[i] = \bot$ holds. In this case, by (I4) and (I38), we have $\mathsf{HistLen}[i] = 0$. Hence, by (I23), the consequent is true before and after the execution of $5.p$.

Statement $7.p$ may establish the antecedent only if executed when $p@\{7\} \wedge p.idx = i$ holds. In this case, by (I7) and (I8), $Signal[i][p.prev]$ is the only entry among $Signal[i][\ldots]$ that is *true*. Thus, statement $7.p$ establishes the consequent.

The only statement that may falsify the consequent is $21.q$, where q is any arbitrary process. Statement $21.q$ may potentially falsify (I5) only if executed when $p@\{8..21\} \wedge p.idx = i \wedge q@\{21\} \wedge q.idx = i$ holds, which implies $p = q$ by (I2). Thus, $21.q$ falsifies the antecedent in this case. $\square$

invariant $Signal[i][x] = true \Rightarrow x = \mathsf{Hist}[i][Position[i]]$ (I6)

Proof: The only statement that may establish the antecedent is $21.p$, which may do so only if $p.idx = i \ \wedge \ p.self = x$ holds. In this case, before $21.p$ is executed, $Position[i] = p.\mathsf{position} + 1$ and $p.self = \mathsf{Hist}[i][p.\mathsf{position} + 1]$ hold, by (I27) and (I3), respectively. Thus, the consequent is true before and after the execution of $21.p$.

The only statements that my falsify the consequent are $5.p$ and $18.p$ (which may update $\mathsf{Hist}[i][Position[i]]$) and $11.p$ (which may update $Position[i]$), where p is any arbitrary process. Statement $5.p$ may update $\mathsf{Hist}[i][Position[i]]$ only if executed when

$$p@\{5\} \ \wedge \ p.idx = i \tag{E.1}$$

holds. In this case, $5.p$ updates only one entry of $\mathsf{Hist}[i][\ldots]$, namely, $\mathsf{Hist}[i][h + 1]$, where h is the value of $\mathsf{HistLen}[i]$ before its execution. Moreover, by (I28), either $h \geq Position[i]$ holds, or there exists a process q satisfying $q@\{19\} \ \wedge \ q.idx = 1 - i$. In the former case, statement $5.p$ does not update $\mathsf{Hist}[i][Position[i]]$, and hence preserves (I5). In the latter case, by (I13), we have $p@\{0..3\} \ \vee \ p.idx = 1 - i$, which contradicts (E.1). It follows that the latter case in fact cannot arise.

Statement $18.p$ may update $\mathsf{Hist}[i][Position[i]]$ only if executed when $p.idx = 1 - i$ holds, in which case, by (I24), the antecedent of (I6) is false before and after its execution. Similarly, statement $11.p$ may update $Position[i]$ only if executed when $p.idx = i$ holds, in which case, by (I5), the antecedent is false before and after its execution. □

invariant $Signal[i][x] = Signal[i][y] = true \ \Rightarrow \ x = y$ (I7)

Proof: This invariant follows trivially from (I6). □

invariant $p@\{7\} \ \wedge \ p.idx = i \ \Rightarrow \ Signal[i][p.prev] = true$ (I8)

Proof: The only statement that may establish the antecedent is $6.p$, which may do so only if the consequent holds.

The only statements that may falsify the consequent are $7.q$ and $17.q$, where q is any arbitrary process. Statement $7.q$ may potentially falsify (I8) only if executed when $p@\{7\} \ \wedge \ p.idx = i \ \wedge \ q@\{7\} \ \wedge \ q.idx = i$ holds. In this case, by (I2), we have $p = q$, and hence $7.q$ falsifies the antecedent. Statement $17.q$ may falsify the consequent only if executed when $q.idx = 1 - i$ holds, which implies that $p@\{0..3\} \ \vee \ p.idx = 1 - i$ holds, by (I13). Thus, the antecedent is false before and after the execution of $17.q$. □

invariant $\left|\{p :: p@\{6,7\} \ \wedge \ p.idx = i \ \wedge \ p.prev = x\}\right| \le 1$ (I9)

Proof: Assume the following.

$$p@\{6,7\} \ \wedge \ q@\{6,7\} \ \wedge \ p.idx = q.idx = i \ \wedge \ p.prev = q.prev = x.$$

Our proof obligation is to show $p = q$. Define $j = p.\mathsf{position}$ and $k = q.\mathsf{position}$. By (I3), we have $(p.prev = \mathsf{Hist}[i][j] = x) \ \wedge \ (j < \mathsf{HistLen}[i])$. Similarly, we also have $(q.prev = \mathsf{Hist}[i][k] = x) \ \wedge \ (k < \mathsf{HistLen}[i])$. Moreover, by (I26) and (I47), we have $j \ge Position[i]$ and $k \ge Position[i]$.

Combining these assertions, by (I48), we have $j = k$. By (I29), this implies $p = q$. □

invariant $Position[i] = q + 1 \ \Rightarrow$
$$q@\{0..3\} \ \vee \ q.idx = i \ \vee \ (\exists p :: p@\{12..20\} \ \wedge \ p.idx = i) \qquad \text{(I10)}$$

Proof: The only statements that may establish the antecedent are $11.r$ and $19.r$, where r is any arbitrary process. Statement $11.r$ may establish the antecedent only if executed when $r.idx = i$ holds, in which case it establishes the last disjunct of the consequent. If $19.r$ updates $Position[i]$, then it establishes $Position[i] = 0$, and hence cannot establish the antecedent. (Recall that q is assumed to range over $0..N-1$.)

The only statement that may falsify $q@\{0..3\} \ \vee \ q.idx = i$ is $3.q$, which may do so only if executed when $CurrentQueue = 1 - i$ holds, which in turn implies that $Position[i] = 0 \ \vee \ Position[i] > N$ holds, by (I37). Thus, since $0 \le q < N$, the antecedent is false before and after $3.q$ is executed.

The only statements that may falsify $p@\{12..20\} \ \wedge \ p.idx = i$ are $13.p$, $14.p$, $15.p$, and $20.p$. Assume that the antecedent holds before the execution of each of these statements. By (I15), we have $p.pos = q < N$, and hence statement $13.p$ establishes $p@\{14\}$, and statement $20.p$ cannot be executed (since $p.pos = N$ holds when it is executed).

Statement $14.p$ may falsify $p@\{12..20\}$ only if executed when $Active[q] = false$ holds. By (I33), this implies that $q@\{0..2\}$ holds, and hence the consequent of (I10) holds after $14.p$ is executed. Statement $15.p$ may falsify $p@\{12..20\} \ \wedge \ p.idx = i$ only if executed when $QueueIdx[q] = i$ holds. By (I34) and (I35), this implies $q@\{0,1\} \ \vee \ q.idx = i$, and hence the consequent of (I10) holds after its execution. □

invariant $q + 1 < Position[i] \le N \ \Rightarrow \ q@\{0..3\} \ \vee \ q.idx = i$ (I11)

Proof: The only statements that may establish the antecedent are 11.r and 19.r, where r is any arbitrary process. Statement 11.r updates $Position[i]$ only if executed when $r.idx = i$ holds, in which case, by (I14), it increments $Position[i]$ by one. Therefore, statement 11.r may establish the antecedent only if executed when $Position[i] = q + 1$ holds. In this case, by (I10), either the consequent holds, or there exists a process p satisfying $p@\{12..20\} \wedge p.idx = i$, before the execution of 11.r. However, since we have $r@\{11\} \wedge r.idx = i$, the latter is precluded by (I2). Therefore, the consequent is true before 11.r is executed, and hence it is also true afterward.

If statement 19.r updates $Position[i]$, then it establishes $Position[i] = 0$, and hence cannot establish the antecedent.

The only statement that may falsify the consequent is 3.q, which may do so only if $CurrentQueue \neq i$ holds. In this case, by (I37), the antecedent is false before and after the execution of 3.q. □

invariant $p@\{11..13\} \wedge p.idx = i \wedge p.pos = N \Rightarrow$

$$CurrentQueue = i \wedge (\forall q :: q@\{0..3\} \vee q.idx = i) \qquad \text{(I12)}$$

Proof: The only statement that may establish the antecedent is 10.p, which may do so only if executed when

$$p@\{10\} \wedge p.idx = i \wedge Position[i] = N \qquad \text{(E.2)}$$

holds. By (I37), this implies that $CurrentQueue = i$ holds before and after 10.p is executed.

In order to prove that $(\forall q :: q@\{0..3\} \vee q.idx = i)$ holds after the execution of 10.p, we consider two cases, depending on the value of q. If $0 \leq q < N - 1$, then by (I11), $q@\{0..3\} \vee q.idx = i$ holds before and after the execution of 10.p. On the other hand, if $q = N-1$, then since $Position[i] = N$, by (I10), either $q@\{0..3\} \vee q.idx = i$ holds, or there exists a process r (different from p) satisfying $r@\{12..20\} \wedge r.idx = i$. However, the latter is precluded by (E.2) and (I2).

The only statement that may falsify $CurrentQueue = i$ is 20.r (where r is any arbitrary process), which may do so only if executed when $r.idx = i$ holds. Taken together with the antecedent, this implies that $r = p$ holds, by (I2). Thus, statement 20.r falsifies the antecedent in this case.

The only statement that may falsify $q@\{0..3\} \vee q.idx = i$ is 3.q. However, if 3.q is executed when the consequent is true, then 3.q establishes $q.idx = i$, and hence preserves the consequent. □

$$\textbf{invariant}\ \ p@\{16..20\} \ \wedge\ p.idx = i \ \Rightarrow \\ \qquad \mathit{CurrentQueue} = i \ \wedge\ (\forall q :: q@\{0..3\} \ \vee\ q.idx = i) \qquad \text{(I13)}$$

Proof: The only statement that may establish the antecedent is $13.p$, which may do so only if executed when $p.idx = i \ \wedge\ p.pos = N$ holds. In this case, by (I12), the consequent holds before and after the execution of statement $13.p$.

The only statements that may falsify the consequent are $3.q$ and $20.q$ (where q is any arbitrary process). However, each preserves the consequent as shown in the proof of (I12). □

$$\textbf{invariant}\ \ p@\{11\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Position}[i] = p.pos \qquad \text{(I14)}$$
$$\textbf{invariant}\ \ p@\{12..21\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Position}[i] = p.pos + 1 \qquad \text{(I15)}$$

Proof: The only statement that may establish the antecedent of (I14) (respectively, (I15)) is $10.p$ (respectively, $11.p$), which clearly establishes the corresponding consequent.

The only other statements that may falsify either consequent are $11.q$ and $19.q$, where q is any arbitrary process. Statement $11.q$ may falsify either consequent only if executed when $q.idx = i$ holds. Taken together with either antecedent, this implies that $q = p$ holds, by (I2). Thus, statement $11.q$ falsifies the antecedent of (I14), and establishes the antecedent and consequent of (I15).

Statement $19.q$ may falsify either consequent only if executed when $q.idx = 1 - i \ \wedge\ q.pos = N$ holds. By (I13), this implies that $p@\{0..3\} \ \vee\ p.idx = 1 - i$ holds. Hence, the antecedents of (I14) and (I15) are false before and after the execution of $19.q$. □

$$\textbf{invariant}\ \ \mathsf{HistLen}[i] > h \ \wedge\ \mathsf{Param}[i][h] = (p, c) \ \Rightarrow \\ \qquad \mathsf{Hist}[i][h+1] = \phi(\mathsf{Hist}[i][h],\ \alpha_p[c]) \qquad \text{(I16)}$$

Proof: The only statement that may establish the antecedent is $5.q$, where q is any arbitrary process. (Note that statement $18.q$ assigns $\mathsf{HistLen}[1 - q.idx] := 0$, and hence cannot establish the antecedent.)

Statement $5.q$ may establish the antecedent only if executed when $\mathsf{HistLen}[i] = h \ \wedge\ q.idx = i$ holds. In this case, by (I17), $5.q$ establishes

$$\mathsf{Hist}[i][h+1] = q.\mathit{self} = \phi(\mathsf{Hist}[i][h],\ \alpha_q[c']) \quad \text{and} \quad \mathsf{Param}[i][h] = (q, c'),$$

where c' is the value of $q.counter$ before the execution of $5.q$. Thus, the antecedent is established only if $q = p$ and $c = c'$, in which case the consequent easily follows.

The only statements that may falsify the consequent are $5.q$ and $18.q$, where q is any arbitrary process. Statement $5.q$ may falsify the consequent only if executed when $q.idx = i \wedge (\mathsf{HistLen}[i] = h-1 \vee \mathsf{HistLen}[i] = h)$ holds. If $\mathsf{HistLen}[i] = h-1$ holds before its execution, then $\mathsf{HistLen}[i] = h$ holds after its execution, and hence the antecedent is false. On the other hand, if $\mathsf{HistLen}[i] = h$ holds before its execution, then statement $5.q$ preserves (I16) as shown above.

Statement $18.q$ may falsify the consequent only if executed when $q.idx = 1-i$ holds, in which case it establishes $\mathsf{HistLen}[i] = 0$, and hence the antecedent is false after its execution. □

invariant $\mathit{Tail}[i] = \mathsf{Hist}[i][\mathsf{HistLen}[i]]$ (I17)
invariant $\mathsf{Hist}[i][0] = \perp$ (I18)

Proof: These invariants follow trivially from inspecting the code. In particular, lines 5c, 5f and 5g, as well as lines 18a and 18d, ensure that (I17) is maintained. Also, since $\mathsf{HistLen}[i]$ is always nonnegative (by (I21)), line 5f cannot update $\mathsf{Hist}[i][0]$, and hence (I18) is maintained. □

invariant $0 \le j < k < \mathsf{HistLen}[i] \;\wedge$
$\mathsf{Param}[i][j] = (p, c_1) \;\wedge\; \mathsf{Param}[i][k] = (p, c_2) \;\wedge$
$(\forall l : j < l < k :: \mathsf{Param}[i][l].proc \ne p) \;\Rightarrow$
$c_2 = c_1 + 1$ (I19)

Proof: The only statement that may establish the antecedent is $5.q$, where q is any arbitrary process. (Note that statement $18.q$ assigns $\mathsf{HistLen}[1-1.idx] := 0$ and does not update any entry of Param.) Since $5.q$ increments $\mathsf{HistLen}[q.idx]$ by one, it may establish the antecedent only if executed when $q.idx = i \;\wedge\; \mathsf{HistLen}[i] = k$ holds. Note that $5.q$ establishes $\mathsf{Param}[i][k] = (q, c)$ in this case, where c is the value of $q.counter$ before the execution of $5.q$. Thus, $5.q$ may establish the antecedent only if executed when the following holds.

$$\begin{aligned}
&q = p \;\wedge\; c_2 = q.counter \;\wedge\; q@\{5\} \;\wedge\; q.idx = i \;\wedge\\
&0 \le j < k = \mathsf{HistLen}[i] \;\wedge\\
&\mathsf{Param}[i][j] = (p, c_1) \;\wedge\\
&(\forall l : j < l < k :: \mathsf{Param}[i][l].proc \ne p).
\end{aligned}$$

Thus, by applying (I20) with 'c' $\leftarrow c_1$, the first disjunct of the consequent of (I20) follows, and hence we have $c_2 = q.counter = c_1 + 1$. □

$$\begin{array}{ll}\textbf{invariant} & 0 \le j < \mathsf{HistLen}[i] \ \wedge \\ & \mathsf{Param}[i][j] = (p, c) \ \wedge \\ & (\forall k : j < k < \mathsf{HistLen}[i] :: \mathsf{Param}[i][k].proc \neq p) \ \Rightarrow \\ & \quad (p.counter = c + 1 \ \wedge \ p@\{4..22\} \ \wedge \ p.idx = i) \ \vee \qquad \mathcal{A} \\ & \quad (p.counter = c + 1 \ \wedge \ p@\{0..3\}) \ \vee \qquad \mathcal{B} \\ & \quad (p@\{0..3\} \ \wedge \ CurrentQueue = 1 - i) \ \vee \qquad \mathcal{C} \\ & \quad (p@\{4..22\} \ \wedge \ p.idx = CurrentQueue = 1 - i) \qquad \mathcal{D} \end{array} \tag{I20}$$

Proof: The only statement that may establish the antecedent is $5.q$, where q is any arbitrary process. (As with (I19), $18.q$ need not be considered here.) Since $5.q$ increments $\mathsf{HistLen}[q.idx]$ by one, it may establish the antecedent only if executed when $q.idx = i \ \wedge \ \mathsf{HistLen}[i] = j$ holds. Note that $5.q$ establishes $\mathsf{Param}[i][k] = (q, c')$, where c' is the value of $q.counter$ before the execution of $5.q$. Thus, the antecedent may be established only if $q = p \ \wedge \ c' = c$ holds. It follows that statement $5.q$ establishes disjunct $\mathcal{A}$ in this case.

Disjunct $\mathcal{A}$ may be falsified only by statements $5.p$ (which may update $p.counter$) and $22.p$ (which may falsify $p@\{4..22\}$). If statement $5.p$ is executed while the antecedent holds, then it establishes $\mathsf{Param}[i][\mathsf{HistLen}[i]-1] = (p, c+1)$, and hence falsifies the last conjunct of the antecedent. Statement $22.p$ establishes disjunct $\mathcal{B}$ if executed when disjunct $\mathcal{A}$ holds.

Disjunct $\mathcal{B}$ may be falsified only by statement $3.p$, which establishes either disjunct $\mathcal{A}$ or disjunct $\mathcal{D}$, depending on the value of *CurrentQueue*.

Disjunct $\mathcal{C}$ may be falsified only by statements $3.p$ and $20.q$, where q is any arbitrary process. Statement $3.p$ establishes disjunct $\mathcal{D}$ if executed when disjunct $\mathcal{C}$ holds. Statement $20.q$ may falsify disjunct $\mathcal{C}$ only if executed when $q.idx = 1 - i$. In this case, by (I39), $\mathsf{HistLen}[i] = 0$ holds before and after the execution of $20.q$, and hence the antecedent of (I20) is false before and after its execution.

Disjunct $\mathcal{D}$ may be falsified only by statements $22.p$ and $20.q$, where q is any arbitrary process. Statement $22.p$ establishes disjunct $\mathcal{C}$ if executed when disjunct $\mathcal{D}$ holds. As shown above, the antecedent is false after the execution of $20.q$. □

$$\textbf{invariant} \ \ 0 \le \mathsf{HistLen}[i] \le 2N \tag{I21}$$

Proof: The only statement that may potentially falsify (I21) is $5.p$, where p is any arbitrary process. Since $5.p$ increments $\mathsf{HistLen}[p.idx]$ by one, it may falsify (I21) only if executed when $p.idx = i \ \wedge\ \mathsf{HistLen}[i] = 2N$ holds. However, this is precluded by (I38). $\square$

invariant $0 \leq Position[i] \leq 2N$ (I22)

Proof: The only statement that may potentially falsify (I22) is $11.p$ (where p is any arbitrary process), which may do so only if executed when $p.idx = i$. In this case, by (I14) and (I26), statement $11.p$ establishes $Position[i] = p.pos + 1 = p.\mathsf{position} + 1$. Moreover, by (I3) and (I21), $p.\mathsf{position} < \mathsf{HistLen}[i] \leq 2N$ holds before $11.p$ is executed. Thus, statement $11.p$ preserves (I22). $\square$

invariant $\mathsf{HistLen}[i] = 0 \ \Rightarrow\ (\forall x :: Signal[i][x] = \mathit{false})$ (I23)

Proof: The only statement that may establish the antecedent is $18.p$, where p is any arbitrary process. Statement $18.p$ may establish the antecedent only if executed when $p@\{18\} \ \wedge\ p.idx = 1 - i$ holds. In this case, by (I24), the consequent holds before and after $18.p$ is executed.

The only statement that may falsify the consequent is $21.p$, where p is any arbitrary process. Statement $21.p$ may falsify the consequent only if executed when $p.idx = i$ holds. In this case, by (I3), $\mathsf{HistLen}[i] > p.\mathsf{position}$ holds before and after $21.p$ is executed. Thus, the antecedent is false before and after the execution of $21.p$. $\square$

invariant $p@\{18\} \ \wedge\ p.idx = i \ \Rightarrow\ (\forall x :: Signal[1 - i][x] = \mathit{false})$ (I24)

Proof: The only statement that may establish the antecedent is $17.p$, which may do so only if $p.idx = i$ holds. Assume that $Signal[1 - i][x] = \mathit{true}$ holds for some x before the execution of $17.p$. It suffices to show $x = p.tail$.

By (I6), we have

$$x = \mathsf{Hist}[1 - i][Position[1 - i]]. \tag{E.3}$$

Also, by (I13), we have

$$(\forall q :: q@\{0..3\} \ \vee\ q.idx = i). \tag{E.4}$$

Moreover, by (I2), $p@\{17\} \ \wedge\ p.idx = i$ implies

$$\neg(\exists r :: r@\{19\} \ \wedge\ r.idx = i). \tag{E.5}$$

Combining (E.4) and (E.5), and applying (I28) with 'i' $\leftarrow 1-i$, we have $\mathsf{HistLen}[1-i] = \mathit{Position}[1-i]$. Hence, by (E.3) and (I17), we have

$$x = \mathsf{Hist}[1-i][\mathsf{HistLen}[1-i]] = \mathit{Tail}[1-i].$$

Thus, by (I25), $x = p.\mathit{tail}$ holds.

The only statement that may falsify the consequent is $21.q$, where q is any arbitrary process. Statement $21.q$ may falsify the consequent only if executed when $q.\mathit{idx} = 1-i$ holds. However, when the antecedent holds, $q@\{21\} \wedge q.\mathit{idx} = 1-i$ is false, by (I13). $\Box$

invariant $p@\{17\} \wedge p.\mathit{idx} = i \Rightarrow p.\mathit{tail} = \mathit{Tail}[1-i]$ (I25)

Proof: The only statement that may establish the antecedent is $16.p$, which may do so only if $p.\mathit{idx} = i$ holds. In this case, $16.p$ establishes the consequent.

The only statements that may falsify the consequent are $5.q$ and $18.q$, where q is any arbitrary process. Statement $5.q$ may falsify the consequent only if executed when $q.\mathit{idx} = 1-i$ holds, which implies that the antecedent is false, by (I13). Similarly, statement $18.q$ may falsify the consequent only if executed when $q.\mathit{idx} = i$ holds, which implies that the antecedent is false, by (I2). $\Box$

invariant $p@\{7..11\} \wedge p.\mathit{idx} = i \Rightarrow \mathit{Position}[i] = p.\mathsf{position}$ (I26)

Proof: The only statements that may establish the antecedent are $5.p$ and $6.p$. Statement $5.p$ may establish the antecedent only if executed when

$$p@\{5\} \wedge \mathit{Tail}[i] = \bot \wedge p.\mathit{idx} = i \tag{E.6}$$

holds. In this case, by (I4) and (I38), we have $\mathsf{HistLen}[i] = 0$. Thus, statement $5.p$ establishes $p.\mathsf{position} = 0$. By (I28), $\mathsf{HistLen}[i] = 0$ also implies that either $\mathit{Position}[i] = 0$ or $(\exists q :: q@\{19\} \wedge q.\mathit{idx} = 1-i)$ holds. In the former case, the consequent is established. In the latter case, by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1-i$, we have $p@\{0..3\} \vee p.\mathit{idx} = 1-i$, which contradicts (E.6). It follows that the latter case in fact cannot arise.

Statement $6.p$ may establish the antecedent only if executed when

$$p@\{6\} \wedge p.\mathit{idx} = i \wedge \mathit{Signal}[i][p.\mathit{prev}] = \mathit{true} \tag{E.7}$$

holds. By (I6), this implies $p.prev = \mathsf{Hist}[i][Position[i]]$. Let $k = p.\mathsf{position}$. By (I3) (with 'h' $\leftarrow k$) and (I47), we have $p.prev = \mathsf{Hist}[i][k]$ and $Position[i] \leq k < \mathsf{HistLen}[i]$. If $k > Position[i]$, then by applying (I48) with 'j' $\leftarrow Position[i]$, we have $\mathsf{Hist}[i][Position[i]] \neq \mathsf{Hist}[i][k]$, a contradiction. It follows that $Position[i] = k = p.\mathsf{position}$ holds before the execution of $6.p$. Thus, it also holds after its execution.

The only statements that may falsify the consequent are $11.q$ and $19.q$, where q is any arbitrary process. Statement $11.q$ may falsify the consequent (when the antecedent holds) only if executed when $q.idx = i$ holds. Taken together with the antecedent, this implies that $q = p$ holds, by (I2). Thus, statement $11.q$ falsifies the antecedent.

Statement $19.q$ may falsify the consequent only if executed when $q.idx = 1-i$ holds. By applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1-i$, this implies $p@\{0..3\} \lor p.idx = 1-i$. Hence, the antecedent is false before and after the execution of $19.q$. □

invariant $p@\{12..21\} \land p.idx = i \Rightarrow Position[i] = p.\mathsf{position} + 1$ (I27)

Proof: The only statement that may establish the antecedent is $11.p$, which may do so only if $p.idx = i$ holds. In this case, by (I14) and (I26), $11.p$ establishes the consequent.

The only statements that may falsify the consequent (while the antecedent holds) are $11.q$ and $19.q$, where q is any arbitrary process. Statement $11.q$ may falsify the consequent only if executed when $q@\{11\} \land q.idx = i$ holds. In this case, if $q = p$, then statement $11.p$ preserves (I27) as shown above. If $q \neq p$, then by (I2), the antecedent is false before and after the execution of $11.q$.

The proof that $19.q$ preserves (I27) is similar to that given in the proof of (I26). □

invariant $\left[\, \mathsf{HistLen}[i] - Position[i] = \left|\{p :: p@\{6..11\} \land p.idx = i\}\right| \,\right] \lor$
$(\exists p :: p@\{19\} \land p.idx = 1-i)$ (I28)

Proof: Define

$$X = \left|\{p :: p@\{6..11\} \land p.idx = i\}\right| .$$

The only statements that may potentially falsify (I28) are $5.q$ (which may modify $\mathsf{HistLen}[i]$ and X), $11.q$ (which may modify $Position[i]$ and X), $18.q$ (which may modify $\mathsf{HistLen}[i]$), and $19.q$ (which may modify $Position[i]$ and also falsify the second disjunct), where q is any arbitrary process.

Statements $5.q$ and $11.q$ may modify $\mathsf{HistLen}[i]$, $Position[i]$, or X only if executed when $q.idx = i$ holds. In this case, $5.q$ increments both $\mathsf{HistLen}[i]$ and X by one, and hence preserves (I28). Similarly, by (I14), $11.q$ increments $\mathit{Position}[i]$ and decrements X by one, and hence preserves (I28).

Statement $18.q$ may update $\mathsf{HistLen}[i]$ only if executed when $p.\mathit{idx} = 1 - i$, in which case it establishes the second disjunct. Statement $19.q$ may falsify the second disjunct only if executed when $q.\mathit{idx} = 1 - i$. In this case, by (I13) and (I39) (with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$), we have $X = 0$ and $\mathsf{HistLen}[i] = 0$, respectively. Since $19.q$ establishes $\mathit{Position}[i] = 0$, it establishes the first disjunct of (I28). $\square$

invariant $\left|\{p :: p@\{6..21\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; p.\mathsf{position} = h\}\right| \leq 1$ (I29)

Proof: Assume that there exists a process p satisfying $p@\{6..21\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; p.\mathsf{position} = h$. By (I3), this implies $\mathsf{HistLen}[i] > h$.

The only statement that may potentially falsify (I29) is $5.q$, where q is any arbitrary process different from p. Statement $5.q$ may falsify (I29) only if executed when $q.\mathit{idx} = i$ holds. However, since $\mathsf{HistLen}[i] > h$, statement $5.q$ establishes $q.\mathsf{position} > h$, and hence cannot increase the left-hand side of (I29). $\square$

invariant $\mathit{CurrentQueue} = i \;\Rightarrow$
$\left|\{p :: p@\{4, 5\} \;\wedge\; p.\mathit{idx} = 1 - i\}\right| \leq 2N - \mathsf{HistLen}[1 - i]$ (I30)

Proof: The only statement that may establish the antecedent is $20.q$, where q is any arbitrary process. Let X denote the value of

$$\left|\{p :: p@\{6..11\} \;\wedge\; p.\mathit{idx} = 1 - i\}\right|$$

prior to the execution of $20.q$. Statement $20.q$ may establish the antecedent only if executed when

$$q@\{20\} \;\wedge\; q.\mathit{idx} = 1 - i \tag{E.8}$$

holds, which also implies the following.

$$\left|\{p :: p@\{6..11, 20\} \;\wedge\; p.\mathit{idx} = 1 - i\}\right| \geq X + 1$$

By (E.8), and by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, $\neg(\exists r :: r@\{19\} \;\wedge\; r.\mathit{idx} = i)$ holds, and hence, by (I28), we have

$$\mathsf{HistLen}[1 - i] - \mathit{Position}[1 - i] = X.$$

Also, since $20.q$ may be executed only if $q.pos = N$ holds, by applying (I15) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, we have

$$Position[1-i] = N + 1.$$

Combining these assertions, we have the following.

$$\begin{aligned} \left|\{p :: p@\{4,5\} \ \wedge \ p.idx = 1-i\}\right| &\leq N - X - 1 \\ &= N - (\mathsf{HistLen}[1-i] - Position[1-i]) - 1 \\ &= N - (\mathsf{HistLen}[1-i] - N - 1) - 1 \\ &= 2N - \mathsf{HistLen}[1-i]. \end{aligned}$$

Thus, the consequent holds before the execution of $20.q$, and hence it also holds after its execution.

The only statements that may falsify the consequent are $3.q$ (which may increment X) and $5.q$ (which may increment $\mathsf{HistLen}[1-i]$), where q is any arbitrary process. If statement $3.q$ is executed while the antecedent holds, then it establishes $q.idx = i$, and hence cannot increment X. Statement $5.q$ may increment $\mathsf{HistLen}[1-i]$ (by one) only if executed when $q.idx = 1 - i$ holds, in which case it also decrements X by one, and hence preserves the consequent. □

invariant $Position[i] = N + 1 \ \Rightarrow$
$\qquad CurrentQueue = 1 - i \ \vee$
$\qquad (\exists p :: p@\{12..20\} \ \wedge \ p.idx = i \ \wedge \ p.pos = N)$ (I31)

Proof: The only statement that may establish the antecedent is $11.q$, where q is any arbitrary process. However, if $11.q$ establishes the antecedent, then by (I14), it also establishes the second disjunct of the consequent.

The only statements that may falsify the consequent are $20.q$ (which may update $CurrentQueue$) and $13.p$, $14.p$, $15.p$, and $20.p$ (which may falsify $p@\{12..20\} \ \wedge \ p.idx = i \ \wedge \ p.pos = N$), where q is any arbitrary process. Statement $20.q$ may falsify the consequent only if executed when $q.idx = 1-i$ holds. In this case, by (I40), $Position[i] = 0$ holds before and after the execution of $20.q$. Thus, the antecedent is false before and after $20.q$ is executed.

Since $p.pos = N$, statement $13.p$ establishes $p@\{16\}$, and statements $14.p$ and $15.p$ cannot be executed. If statement $20.p$ is executed when $p.idx = i$ holds, then it establishes $CurrentQueue = 1 - i$, and hence preserves the consequent. □

invariant $\mathit{Position}[i] > N + 1 \;\Rightarrow\; \mathit{CurrentQueue} = 1 - i$ (I32)

Proof: The only statement that may establish the antecedent is $11.p$, where p is any arbitrary process. By (I14), $11.p$ may establish the antecedent only if executed when $p@\{11\} \wedge p.\mathit{idx} = i \wedge \mathit{Position}[i] = N + 1$ holds. In this case, by (I2) and (I31), we have the consequent.

The only statement that may falsify the consequent is $20.q$, where q is any arbitrary process. As shown in the proof of (I31), if $20.q$ falsifies the consequent, then the antecedent is false after its execution. □

invariant $\mathit{Active}[p] = p@\{3..22\}$ (I33)
invariant $p@\{5..22\} \wedge p.\mathit{idx} = i \;\Rightarrow\; \mathit{QueueIdx}[p] = i$ (I34)
invariant $p@\{2..4\} \;\Rightarrow\; \mathit{QueueIdx}[p] = \bot$ (I35)

Proof: These invariants follow trivially from inspecting Algorithm G-CC. □

invariant $\mathit{Position}[i] \leq N \;\Rightarrow$

$$\begin{aligned} & \mathit{CurrentQueue} = i \;\vee && \mathcal{A} \\ & (\exists p :: p@\{20\} \wedge p.\mathit{idx} = 1 - i) \;\vee && \mathcal{B} \\ & \big(\mathit{Position}[i] = 0 \wedge (\forall q :: q@\{0..3\} \vee q.\mathit{idx} = 1 - i)\big) && \mathcal{C} \end{aligned}$$
(I36)

Proof: The only statements that may establish the antecedent are $11.r$ and $19.r$, where r is any arbitrary process. However, if statement $11.r$ updates $\mathit{Position}[i]$, then by (I14), it increments $\mathit{Position}[i]$ by one. It follows that, although statement $11.r$ may preserve the antecedent, it cannot establish it. Statement $19.r$ may establish the antecedent only if executed when $r.\mathit{idx} = 1 - i$ holds, in which case it establishes disjunct $\mathcal{B}$.

The only statement that may falsify disjunct $\mathcal{A}$ is $20.r$, where r is any arbitrary process. Statement $20.r$ may falsify disjunct $\mathcal{A}$ only if executed when $r.\mathit{idx} = i \wedge r.\mathit{pos} = N$ holds, which implies that $\mathit{Position}[i] = N + 1$ holds, by (I15). Thus, in this case, the antecedent is false before and after the execution of $20.r$.

The only statement that may falsify disjunct $\mathcal{B}$ is $20.p$, which establishes disjunct $\mathcal{A}$.

The only statements that may falsify disjunct $\mathcal{C}$ are $3.q$ and $11.q$, where q is any arbitrary process. Statement $3.q$ may falsify disjunct $\mathcal{C}$ only if executed when $\mathit{CurrentQueue} = i$ holds, in which case disjunct $\mathcal{A}$ holds before and after its execution. Statement $11.q$ may falsify disjunct $\mathcal{C}$ only if executed when $q@\{11\} \wedge q.\mathit{idx} = i$ holds, which is precluded when disjunct $\mathcal{C}$ holds. □

invariant $1 \leq \mathit{Position}[i] \leq N \;\Rightarrow\; \mathit{CurrentQueue} = i$ (I37)

Proof: By (I36), the antecedent implies one of the following.

$$\begin{aligned}
\mathcal{A}: &\quad \mathit{CurrentQueue} = i, \\
\mathcal{B}: &\quad (\exists p :: p@\{20\} \,\wedge\, p.\mathit{idx} = 1 - i), \quad \text{or} \\
\mathcal{C}: &\quad \mathit{Position}[i] = 0 \,\wedge\, (\forall q :: q@\{0..3\} \,\vee\, q.\mathit{idx} = 1 - i).
\end{aligned}$$

By (I40), $\mathcal{B}$ implies $\mathit{Position}[i] = 0$. Also, $\mathcal{C}$ clearly implies $\mathit{Position}[i] = 0$. Thus, both are precluded by the antecedent. It follows that $\mathcal{A}$ is true. $\Box$

invariant $p@\{4, 5\} \,\wedge\, p.\mathit{idx} = i \;\Rightarrow\; \mathsf{HistLen}[i] < 2N$ (I38)

Proof: For the sake of contradiction, assume

$$p@\{4, 5\} \,\wedge\, p.\mathit{idx} = i \,\wedge\, \mathsf{HistLen}[i] \geq 2N. \tag{E.9}$$

By applying (I30) with 'i' $\leftarrow 1 - i$, we have $\mathit{CurrentQueue} \neq 1 - i$, *i.e.*,

$$\mathit{CurrentQueue} = i. \tag{E.10}$$

Thus, by (I32), we have

$$\mathit{Position}[i] \leq N + 1. \tag{E.11}$$

Also, (E.9) implies

$$\big|\{q :: q@\{6..11\} \,\wedge\, q.\mathit{idx} = i\}\big| \leq N - 1.$$

Hence, by (I28), we have

$$\mathsf{HistLen}[i] - \mathit{Position}[i] \leq N - 1 \quad \vee \quad (\exists r :: r@\{19\} \,\wedge\, r.\mathit{idx} = 1 - i).$$

However, if there exists a process r satisfying $r@\{19\} \wedge r.\mathit{idx} = 1 - i$, then by (I13) (with '$p$' $\leftarrow r$ and 'i' $\leftarrow 1 - i$), we have $p@\{0..3\} \,\vee\, p.\mathit{idx} = 1 - i$, which contradicts (E.9). Therefore, we have $\mathsf{HistLen}[i] - \mathit{Position}[i] \leq N - 1$.

Note that the only common solution to $\mathsf{HistLen}[i] \geq 2N$ (given in (E.9)), (E.11), and $\mathsf{HistLen}[i] - \mathit{Position}[i] \leq N - 1$ is $\mathsf{HistLen}[i] = 2N$ and

$$\mathit{Position}[i] = N + 1.$$

By (E.10) and (I31), this implies that $(\exists r :: r@\{12..20\} \ \wedge \ r.idx = i)$ holds. From this and (E.9), we have

$$\left|\{q :: q@\{6..11\} \ \wedge \ q.idx = i\}\right| \leq N - 2.$$

Hence, by (I28), we have

$$\mathsf{HistLen}[i] - Position[i] \leq N - 2 \quad \vee \quad (\exists r :: r@\{19\} \ \wedge \ r.idx = 1 - i).$$

The second disjunct is precluded by (I13) as shown above, and hence we have $\mathsf{HistLen}[i] - Position[i] \leq N - 2$. However, this cannot hold simultaneously with (E.9) and (E.11). Thus, we have reached a contradiction. $\square$

invariant $p@\{19, 20\} \ \wedge \ p.idx = i \ \Rightarrow \ \mathsf{HistLen}[1 - i] = 0$ (I39)

Proof: The antecedent may be established only by statement 18.p, which may do so only if $p.idx = i$ holds. In this case, 18.p also establishes the consequent.

The only statement that may falsify the consequent is 5.q, where q is any arbitrary process. Statement 5.q may potentially falsify (I39) only if executed when

$$p@\{19, 20\} \ \wedge \ p.idx = i \ \wedge \ q@\{5\} \ \wedge \ q.idx = 1 - i$$

holds. However, this contradicts (I13). $\square$

invariant $p@\{20\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[1 - i] = 0$ (I40)

Proof: The antecedent may be established only by statement 19.p, which may do so only if $p.idx = i$ holds. In this case, 19.p also establishes the consequent.

The only statement that may falsify the consequent is 11.q, where q is any arbitrary process. Statement 11.q may potentially falsify (I40) only if executed when $p@\{20\} \ \wedge \ p.idx = i \ \wedge \ q@\{11\} \ \wedge \ q.idx = 1 - i$ holds. However, this contradicts (I13). $\square$

invariant $Position[i] \leq h < \mathsf{HistLen}[i] \ \Rightarrow$
$(\exists p :: p@\{6..11\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h)$ (I41)

Proof: The only statements that may establish the antecedent are 5.q and 18.q (which may modify $\mathsf{HistLen}[i]$) and 11.q and 19.q (which may modify $Position[i]$), where q is any arbitrary process. Statement 5.q may establish $\mathsf{HistLen}[i] > h$ only if executed when $q.idx = i \ \wedge\ \mathsf{HistLen}[i] = h$ holds, in which case it establishes the consequent.

If statement 18.q modifies $\mathsf{HistLen}[i]$, then it establishes $\mathsf{HistLen}[i] = 0$, and hence falsifies the antecedent.

Statement 11.q may modify $Position[i]$ only if executed when $q.idx = i$ holds. In this case, by (I14), it increments $Position[i]$ by one. Hence, although 11.q may preserve the antecedent, it cannot establish it.

Statement 19.q may establish $Position[i] \le h$ only if executed when $q.idx = 1 - i$ holds, in which case, by (I39), it establishes $Position[i] = \mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

The only statement that may falsify the consequent is 11.p, which may do so only if executed when $p.idx = i \ \wedge\ p.\mathsf{position} = h$ holds. In this case, by (I14) and (I26), statement 11.p establishes $Position[i] = h + 1$, and hence falsifies the antecedent. □

invariant $Position[i] = h > 0 \ \Rightarrow$

$$\begin{array}{lr} Signal[i][\mathsf{Hist}[i][h]] = true \ \vee & \mathcal{A} \\ \big(\exists p :: p@\{12..21\} \ \wedge\ p.idx = i \ \wedge\ p.\mathsf{position} = h - 1\big) \ \vee & \mathcal{B} \\ \big(\exists q :: q@\{8..11\} \ \wedge\ q.idx = i \ \wedge\ q.\mathsf{position} = h\big) & \mathcal{C} \end{array} \tag{I42}$$

Proof: The only statement that may establish the antecedent is 11.p, where p is any arbitrary process. By (I14) and (I26), statement 11.p may establish the antecedent only if executed when $p.idx = i \ \wedge\ p.\mathsf{position} = h - 1$ holds, in which case it establishes disjunct $\mathcal{B}$.

The only statements that may falsify disjunct $\mathcal{A}$ are 5.p, 7.p, and 18.p, where p is any arbitrary process. Statement 5.p may falsify disjunct $\mathcal{A}$ only if executed when

$$p@\{5\} \ \wedge\ p.idx = i \ \wedge\ \mathsf{HistLen}[i] = h - 1 \tag{E.12}$$

holds. Combining this with the antecedent, and using (I28), this implies $(\exists q :: q@\{19\} \wedge q.idx = 1 - i)$. Hence, by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, we have $p@\{0..3\} \ \vee\ p.idx = 1 - i$, which contradicts (E.12). It follows that statement 5.p cannot falsify disjunct $\mathcal{A}$ while the antecedent holds.

Statement $7.p$ may falsify disjunct $\mathcal{A}$ only if executed when $p.idx = i$ holds, in which case, by (I26), we have $p.\mathsf{position} = Position[i] = h$. Hence, statement $7.p$ establishes disjunct $\mathcal{C}$ in this case.

Statement $18.p$ may falsify disjunct $\mathcal{A}$ only if executed when $p.idx = 1 - i$ holds. In this case, by (I24), disjunct $\mathcal{A}$ is already false before $18.p$ is executed.

The only statement that may falsify disjunct $\mathcal{B}$ is $21.p$ (where p is any arbitrary process), which may do so only if executed when $p.idx = i \ \wedge\ p.\mathsf{position} = h - 1$ holds. In this case, by (I3), $p.self = \mathsf{Hist}[i][h]$ holds before its execution. Thus, statement $21.p$ establishes disjunct $\mathcal{A}$.

The only statement that may falsify disjunct $\mathcal{C}$ is $11.p$ (where p is any arbitrary process), which may do so only if executed when $p.idx = i \ \wedge\ p.\mathsf{position} = h$ holds. In this case, by (I14) and (I26), $11.p$ establishes $Position[i] = h + 1$, and hence falsifies the antecedent. □

invariant $0 < j < k \leq \mathsf{HistLen}[i] \ \wedge\ k < 2N \ \wedge$
$\mathsf{Param}[i][j-1].proc \neq \mathsf{Param}[i][k-1].proc \ \Rightarrow$
$\mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I43)

invariant $0 < j \leq \mathsf{HistLen}[i] \ \wedge\ j < 2N \ \Rightarrow\ \mathsf{Hist}[i][j] \neq \bot$ (I44)

invariant $0 < j < k \leq \mathsf{HistLen}[i] \ \wedge\ k < 2N \ \wedge$
$\mathsf{Param}[i][j-1] = (p, c) \ \wedge\ \mathsf{Param}[i][k-1] = (p, c+1) \ \Rightarrow$
$\mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I45)

Proof: Invariants (I43)–(I45) follow easily from invariants (I16), (I17), (I18), and (I19), together with the assumption that the underlying *fetch-and-ϕ* primitive has rank at least $2N$. In particular, (I43) states that any two invocations (among the first $2N - 1$) by different processes write different values to $Tail[i]$; (I44) states that each of the first $2N - 1$ invocations writes to $Tail[i]$ a value different from $\bot$; (I45) states that any two successive invocations (among the first $2N - 1$) by the same process write different values to $Tail[i]$. □

invariant $Position[i] \leq h < \mathsf{HistLen}[i] \ \wedge\ \mathsf{Param}[i][h].proc = p \ \Rightarrow$
$p@\{6..11\} \ \wedge\ p.idx = i \ \wedge\ p.\mathsf{position} = h$ (I46)

Proof: The only statements that may establish the antecedent are $5.q$ and $18.q$ (which may update $\mathsf{HistLen}[i]$ and $\mathsf{Param}[i][h].proc$) and $11.q$ and $19.q$ (which may update $Position[i]$), where q is any arbitrary process.

Statement $5.q$ may establish $h < \mathsf{HistLen}[i] \ \wedge \ \mathsf{Param}[i][h].proc = p$ only if executed when $\mathsf{HistLen}[i] = h \ \wedge \ p = q \ \wedge \ q.idx = i$ holds, in which case it establishes the antecedent.

Statement $18.q$ may update $\mathsf{HistLen}[i]$ or $\mathsf{Param}[i][h]$ only if $q.idx = 1 - i$, in which case it establishes $\mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

Statement $11.q$ may update $Position[i]$ only if executed when $q.idx = i$ holds. In this case, by (I14), it increments $Position[i]$ by one. Hence, although $11.q$ may preserve the antecedent, it cannot establish it.

Statement $19.q$ may establish $Position[i] \leq h$ only if executed when $q.idx = 1 - i$ holds, in which case, by (I39), it establishes $Position[i] = \mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

The only statement that may falsify the consequent is $11.p$, which may do so only if executed when $p.idx = i \ \wedge \ p.\mathsf{position} = h$ holds. In this case, by (I14) and (I26), statement $11.p$ establishes $Position[i] = h + 1$, and hence the antecedent is false after its execution. □

invariant $p@\{6\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] \leq p.\mathsf{position}$ (I47)

Proof: The only statement that may establish the antecedent is $5.p$, which may do so only if $p.idx = i$. Let h be the value of $\mathsf{HistLen}[i]$ before the execution of $5.p$. By (I28), we have either $Position[i] \leq h$ or $(\exists q :: q@\{19\} \ \wedge \ q.idx = 1-i)$. However, by applying (I13) with 'i' $\leftarrow 1 - i$, the latter implies $p@\{0..3\} \ \vee \ p.idx = 1 - i$, which implies that the antecedent is false. On the other hand, if $Position[i] \leq h$ holds before the execution of $5.p$, then $Position[i] \leq h = p.\mathsf{position}$ is established.

The only statement that may falsify the consequent (while the antecedent holds) is $11.q$ (where q is any arbitrary process), which may do so only if $q.idx = i$. In this case, by (I14), $11.q$ increments $Position[i]$ by one, and hence it may falsify the consequent only if executed when $Position[i] = p.\mathsf{position}$ holds. By applying (I26) with 'p' $\leftarrow q$, we also have $Position[i] = q.\mathsf{position}$. Combining these assertions with the antecedent, and using (I29), we have $p = q$. However, in this case, the antecedent is false after the execution of $11.q$. □

invariant $Position[i] \leq j < k < \mathsf{HistLen}[i] \ \Rightarrow \ \mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I48)

Proof: The only statements that may establish the antecedent are $5.p$ and $18.p$ (which may update $\mathsf{HistLen}[i]$, $\mathsf{Hist}[i][j]$, or $\mathsf{Hist}[i][k]$) and $11.p$ and $19.p$ (which may update $Position[i]$), where p is any arbitrary process.

Statement $5.p$ may establish the antecedent only if executed when $p.idx = i \ \wedge$ $Position[i] \leq j < k = \mathsf{HistLen}[i]$ holds. In this case, by (I38),

$$Position[i] \leq j < k = \mathsf{HistLen}[i] < 2N \tag{E.13}$$

holds before its execution. We consider two cases.

- If $j = 0$, then by (I18), we have $\mathsf{Hist}[i][j] = \bot$. Also, by applying (I44) with 'j' $\leftarrow k$, and using (E.13), we have $\mathsf{Hist}[i][k] \neq \bot$. Hence, the consequent of (I48) holds before and after the execution of $5.p$.

- If $j > 0$, then let (q, c_1) denote the value of $\mathsf{Param}[i][j-1]$ and (r, c_2) denote the value of $\mathsf{Param}[i][k-1]$ prior to the execution of $5.p$. If $q \neq r$, then by (E.13) and (I43), the consequent holds before and after the execution of $5.p$.

  Therefore, assume that $q = r$. Let q_l denote the value of $\mathsf{Param}[i][l-1].proc$ prior to the execution of $5.p$, for each $0 < l \leq k$. Then, we have $q = q_j = q_k$.

  For each l satisfying $Position[i] < l \leq k$, by applying (I46) with 'p' $\leftarrow q_l$ and 'h' $\leftarrow l - 1$, and using (E.13), we have $q_l.\mathsf{position} = l - 1$ prior to the execution of $5.p$. In particular, we have $q.\mathsf{position} = q_k.\mathsf{position} = k - 1$, and

$$(\forall l : Position[i] < l < k :: q_l \neq q). \tag{E.14}$$

  Since $q_j = q$, this implies that $Position[i] < j < k$ is false. Thus, by (E.13), we have $j = Position[i]$. Combining this with (E.14), we also have

$$(\forall l : j < l < k :: \mathsf{Param}[i][l-1].proc \neq q)$$

  prior to the execution of $5.p$.

  Therefore, by applying (I19) with 'p' $\leftarrow q$, 'j' $\leftarrow j-1$, and 'k' $\leftarrow k-1$, and using (E.13) and the assertions above, we have $c_2 = c_1 + 1$. Combining this with (E.13), and using (I45), it follows that the consequent holds both before and after the execution of $5.p$.

If statement $18.p$ updates $\mathsf{HistLen}[i]$, then it establishes $\mathsf{HistLen}[i] = 0$, and hence the antecedent is false after its execution.

If statement $11.p$ updates $Position[i]$, then by (I14), it increments $Position[i]$ by one. Hence, although $11.p$ may preserve the antecedent, it cannot establish it.

Statement $19.p$ may establish $Position[i] \leq j$ only if executed when $q.idx = 1 - i$ holds, in which case, by (I39), it establishes $Position[i] = \mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

The only statement that may falsify the consequent is $5.p$ (which may update either $\mathsf{Hist}[i][j]$ or $\mathsf{Hist}[i][k]$), where p is any arbitrary process. Statement $5.p$ may update $\mathsf{Hist}[i][j]$ only if executed when $p.idx = i \ \wedge \ \mathsf{HistLen}[i] = j - 1$ holds, in which case it establishes $\mathsf{HistLen}[i] = j$. Thus, in this case, the antecedent is false after $5.p$ is executed. Similar reasoning applies to $\mathsf{Hist}[i][k]$. □

E.3 Proof of Starvation-freedom

We begin with proving the following *unless* and *leads-to* properties. (A *leads-to* B is true if and only if the following holds: if A holds at some state, then eventually B holds at either the same state or some later state. *Leads-to* properties must hold only in fair histories. Recall that A *unless* B is true if and only if the following holds: if $A \wedge \neg B$ holds before some statement execution, then $A \vee B$ holds after that execution. Informally, A is not falsified until B is established.) Informally, (U1) states that if a process p is waiting at statement 6, and if the busy-waiting condition is established, then it holds continuously until p exits the busy-waiting loop. (L1) (respectively, (L2)) is used to prove that, if p has entered the current queue (respectively, the old queue), and waits at statement 6, then the busy-waiting condition is eventually established. Throughout this section, we assume that the $\texttt{Entry}_2$ and $\texttt{Exit}_2$ routines are starvation-free.

$$p@\{6\} \ \wedge \ p.idx = i \ \wedge \ Signal[i][p.prev] = true \quad unless \quad p@\{7\} \tag{U1}$$

Proof: The only statement that may falsify $p@\{6\} \wedge p.idx = i$ is $6.p$, which establishes $p@\{7\}$.

$Signal[i][p.prev] = true$ may be falsified only if some process q $(\neq p)$ executes statement $7.q$ when $q.idx = i \ \wedge \ q.prev = p.prev$ holds. However, if the left-hand side of (U1) is true, then this is precluded by (I9). □

$$\begin{aligned} &CurrentQueue = i \ \wedge \ p@\{6\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h \ \wedge \ Position[i] = k \\ &\qquad leads\text{-}to \ \ (Signal[i][p.prev] = true \ \vee \ Position[i] = k + 1) \end{aligned} \tag{L1}$$

$$\begin{aligned} &CurrentQueue = 1 - i \ \wedge \ p@\{6\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h \ \wedge \ Position[i] = k \\ &\qquad leads\text{-}to \ \ (Signal[i][p.prev] = true \ \vee \ Position[i] = k + 1) \end{aligned} \tag{L2}$$

Proof: We prove (L1) and (L2) by induction on h. Since their proofs are nearly identical, we simply say the "left-hand side" when the argument applies to both (L1) and (L2).

First, assume $h = 0$. The assertion $p@\{6\} \wedge p.idx = i \wedge p.\mathsf{position} = h$ may be established only if statement $5.p$ is executed when $\mathsf{HistLen}[i] = 0$ holds. However, by (I17) and (I18), this implies that $Tail[i] = \mathsf{Hist}[i][0] = \bot$ holds. Thus, statement $5.p$ establishes $p@\{8\}$, and cannot establish the left-hand side. It follows that (L1) and (L2) hold vacuously for $h = 0$.

Now assume that $h > 0$, and that (L1) and (L2) hold for smaller values of h. Consider a state t that satisfies the left-hand side.

By (I13), the left-hand side implies $\neg(\exists q :: q@\{19\} \wedge q.idx = 1-i)$. Thus, by (I28), and using $p@\{6\} \wedge p.idx = i$, we have $k < \mathsf{HistLen}[i]$. By applying (I41) with 'h' $\leftarrow k$, this in turn implies that a process q exists such that

$$q@\{6..11\} \wedge q.idx = i \wedge q.\mathsf{position} = Position[i] = k. \tag{E.15}$$

We consider two cases, depending on the value of k.

Case 1: $\boldsymbol{k = 0}$. If $k = 0$, then by (I18), and by applying (I3) with 'p' $\leftarrow q$ and 'h' $\leftarrow k$, we have $q.prev = \mathsf{Hist}[i][0] = \bot$. Because $q@\{6, 7\} \Rightarrow q.prev \neq \bot$ is (trivially) an invariant, this implies that $q@\{8..11\}$ holds. Thus, q eventually executes statement 11 while $q.idx = i \wedge q.\mathsf{position} = k$ holds. In this case, by (I14) and (I26), $11.q$ establishes $Position[i] = k + 1$, and hence the right-hand side of (L1)/(L2) is established.

Case 2: $\boldsymbol{k > 0}$. Let x be the value of $\mathsf{Hist}[i][k]$ at state t. By (I42), (E.15) implies that one of the following holds at state t, where r is some process.

$$\begin{aligned}
&\mathcal{A}: && Signal[i][x] = true, \\
&\mathcal{B}: && r@\{12..21\} \wedge r.idx = i \wedge r.\mathsf{position} = k - 1, \quad \text{or} \\
&\mathcal{C}: && r@\{8..11\} \wedge r.idx = i \wedge r.\mathsf{position} = k.
\end{aligned}$$

Moreover, by (I3), $q.prev = x$ also holds at state t. We now prove that, in each of the three cases given by $\mathcal{A}$–$\mathcal{C}$, the right-hand side of (L1)/(L2) is eventually established. Toward this goal, we prove the following three claims.

Claim 1: If $\mathcal{A}$ is true at state u, where u is either t or some later state, then the right-hand side of (L1)/(L2) is true at either t or some later state.

Claim 2: If $\mathcal{B}$ is true at state t, then $\mathcal{A}$ is true at either t or some later state u.

Claim 3: If $\mathcal{C}$ is true at state t, then the right-hand side of (L1)/(L2) is true at either t or some later state.

Proof of Claim 1: First, if (E.15) is false at state u, then since it holds at state t, the execution of statement $11.q$ occurs between state t and state u. Note that $q.idx$ and $q.\mathsf{position}$ do not change while $q@\{6..11\}$ holds. Hence, by (I14) and (I26), $q.\mathsf{position} = q.pos = Position[i] = k$ holds before the execution of $11.q$. Therefore, $11.q$ establishes the right-hand side of (L1)/(L2).

On the other hand, assume that (E.15) is true at state u. Then, by (I3), $q.prev = x$ holds at state u. Moreover, by (U1), if $q@\{6\} \land q.prev = x$ holds at state u, then $\mathcal{A}$ continues to hold until $q@\{7\}$ is established. It follows that q eventually executes statement $11.q$, which establishes the right-hand side of (L1)/(L2) as shown above. $\square$

Proof of Claim 2: Assume that $\mathcal{B}$ holds at state t. In this case, by (I3), $r.self = \mathsf{Hist}[i][k] = x$ holds at state t. Hence, if r eventually executes statement 21, then $21.r$ establishes $\mathcal{A}$.

Thus, it suffices to show that statement $21.r$ is eventually executed. Since $r@\{12..21\}$ holds at state t, it suffices to show the following.

> *If $r@\{14, 15\}$ holds at state u, where u is either t or some later state, then $r@\{21\}$ is eventually established.*

Clearly, $r@\{14, 15\}$ implies $0 \le r.pos < N$, which is true at state t (where $\mathcal{B}$ holds) as well as state u. Thus, by (I15), $1 \le Position[i] = r.pos + 1 \le N$ holds at state t. Hence, by (I37), $CurrentQueue = i$ holds at state t. Thus, if the left-hand side of (L2) is true at state t, then $\mathcal{B}$ is false at state t.

On the other hand, if the left-hand side of (L1) is true at state t, then by (L7), given later, $r@\{21\}$ is eventually established. (Note that the proof of (L2) does not depend on (L7). As explained shortly, this is necessary in order to avoid circular reasoning.) $\square$

Proof of Claim 3: Clearly, r eventually executes statement $11.r$ if $\mathcal{C}$ holds. Thus, by (I14) and (I26), $11.r$ establishes $Position[i] = k+1$, and hence the right-hand side of (L1)/(L2) is established. $\square$

Finally, from these three claims, (L1) and (L2) follow. $\square$

The reader may wonder why we have two separate properties (L1) and (L2), when they can be proved in essentially the same way. The reason is that the proof of (L7), given later, indirectly depends on (L2). Since the proof of (L1) depends on (L7), (L1) and (L2) must be kept separate to avoid circular reasoning.

The following properties are consequences of (U1), (L1), and (L2).

$$CurrentQueue = i \ \wedge\ p@\{6\} \ \wedge\ p.idx = i \quad \textit{leads-to} \quad p@\{7\} \tag{L3}$$

$$CurrentQueue = 1 - i \ \wedge\ p@\{6\} \ \wedge\ p.idx = i \quad \textit{leads-to} \quad p@\{7\} \tag{L4}$$

Proof: Since $Position[i]$ is bounded by (I22), by inductively applying (L1) and (L2), respectively, we have the following.

$$\begin{aligned} &CurrentQueue = i \ \wedge\ p@\{6\} \ \wedge\ p.idx = i \ \ \textit{leads-to} \\ &\qquad Signal[i][p.prev] = true; \end{aligned} \tag{E.16}$$

$$\begin{aligned} &CurrentQueue = 1 - i \ \wedge\ p@\{6\} \ \wedge\ p.idx = i \ \ \textit{leads-to} \\ &\qquad Signal[i][p.prev] = true. \end{aligned} \tag{E.17}$$

Assume that the left-hand side of (L3) holds at some state t. By (E.16), $Signal[i][p.prev] = true$ is eventually established at some later state u. If $p@\{7\}$ is established before state u, then (L3) holds. Otherwise, $p@\{6\}$ continues to hold from state t to u. Thus, $p.idx = i$ also continues to hold from state t to u. Therefore, the left-hand side of (U1) holds at state u. By (U1), $Signal[i][p.prev] = true$ is not falsified until $p@\{7\}$ is established, and hence p eventually establishes $p@\{7\}$ by executing statement 6.

The reasoning for (L4) is similar, except that (E.17) is used instead of (E.16). $\square$

Note that the proof of (L3) indirectly depends on (L7), while the proof of (L4) does not.

The following properties state that, if a process p is waiting for process q at statements 14 and 15, then the busy-waiting condition is eventually established. (Note that $q@\{0\}$ implies $Active[q] = false$ by (I33), and $q@\{6\} \wedge q.idx = i$ implies $QueueIdx[q] = i$ by (I34).)

$p@\{14,15\} \;\wedge\; p.idx = i \;\wedge\; p.pos = q$
$\quad$ *leads-to* $q@\{0\} \;\vee\; (q@\{6\} \;\wedge\; q.idx = i) \;\vee\; p@\{21\}$ (L5)

$p@\{14,15\} \;\wedge\; p.idx = i \;\wedge\; p.pos = q \;\wedge\; q@\{1\}$
$\quad$ *leads-to* $(q@\{6\} \;\wedge\; q.idx = i) \;\vee\; p@\{21\}$ (L6)

Proof: Assume that the left-hand side of either (L5) or (L6) holds at state t. By (I2), one of the following holds at t.

$$\begin{aligned}
\mathcal{A}:&\quad q@\{3..22\} \;\wedge\; q.idx = 1 - i;\\
\mathcal{B}:&\quad q@\{22\} \;\wedge\; q.idx = i;\\
\mathcal{C}:&\quad q@\{0\};\\
\mathcal{D}:&\quad q@\{1..3\};\\
\mathcal{E}:&\quad q@\{4,5\} \;\wedge\; q.idx = i;\\
\mathcal{F}:&\quad q@\{6\} \;\wedge\; q.idx = i.
\end{aligned}$$

Also, by (I15), $Position[i] = q + 1$ holds at state t, and hence, by (I37), $CurrentQueue = i$ also holds at state t.

If $p@\{21\}$ is established at some future state, then (L5) and (L6) both hold. Thus, in the rest of the proof, we assume that $p@\{14,15\} \;\wedge\; p.idx = i$ holds continuously at and after state t. We claim that $CurrentQueue = i$ also holds at all future states. Note that $CurrentQueue = i$ may be falsified only by statement $20.r$ (where r is any arbitrary process), which may do so only if executed when $r.idx = i$ holds. However, by (I2), this is precluded when $p@\{14,15\} \;\wedge\; p.idx = i$ holds. Thus, we have the following.

- $p@\{14,15\} \;\wedge\; p.idx = i \;\wedge\; CurrentQueue = i$ holds at t and all later states. (E.18)

Note that the left-hand side of (L6) implies $\mathcal{D}$, and $\mathcal{F}$ implies the right-hand side of (L6). Hence, in order to prove (L6), it suffices to prove the following.

- If $\mathcal{D} \vee \mathcal{E}$ holds at state u, where u is either t or some later state, then $\mathcal{F}$ is established at some state after u. (E.19)

Also, since $\mathcal{C} \;\vee\; \mathcal{F}$ implies the right-hand side of (L5), in order to prove (L5), it suffices to prove the following claim in addition to (E.19).

- If $\mathcal{A} \vee \mathcal{B}$ holds at state u, where u is either t or some later state, then $\mathcal{C}$ is established at some state after u. (E.20)

We prove (E.19) and (E.20) by considering each of $\mathcal{A}$, $\mathcal{B}$, $\mathcal{D}$, and $\mathcal{E}$.

- Assume that $\mathcal{A}$ holds at state u. We claim that, in this case, $\mathcal{B}$ is eventually established.

  It suffices to show that the busy-waiting loops at statement 6 and statements 14 and 15 eventually terminate for q. By applying (L4) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1-i$, and using $CurrentQueue = i$ (given in (E.18)), it follows that the former loop eventually terminates. If $q@\{14, 15\}$, then by (I15), we have $Position[1-i] = q.pos+1$. Also, by (E.18) and (I37), we have $Position[1-i] = 0 \vee Position[1-i] > N$. Combining these two assertions, we have $q.pos = Position[1-i] - 1 \geq N$, and hence $q@\{14, 15\}$ is false. It follows that q in fact does not execute the busy-waiting loop at statements 14 and 15 while $\mathcal{A}$ holds.

- Assume that $\mathcal{B}$ holds at state u. Clearly, $\mathcal{C}$ is eventually established.

- Assume that $\mathcal{D}$ holds at state u. Then, by (E.18), $\mathcal{E}$ is eventually established.

- Assume that $\mathcal{E}$ holds at state u. In this case, q eventually executes statement 5. By (I38), $\mathsf{HistLen}[i] < 2N$ holds before the execution of $5.q$. Moreover, by (E.18) and (I3), $\mathsf{HistLen}[i] > p.\mathsf{position} \geq 0$ also holds. (Note that $\mathsf{HistLen}[i]$ is always nonnegative by (I21). Since $p.\mathsf{position}$ is updated only by line **5d**, it follows that $p.\mathsf{position}$ is always nonnegative.) Combining these two assertions with (I4), we have $Tail[i] \neq \perp$. It follows that statement $5.q$ establishes $\mathcal{F}$.

From the reasoning above, assertions (E.20) and (E.19) follow. Therefore, we have (L5) and (L6). $\square$

Note that the proofs of (L5) and (L6) do not depend on (L7).

The following property states that the busy-waiting loop at statements 14 and 15 eventually terminates.

$$p@\{14, 15\} \quad \textit{leads-to} \quad p@\{21\} \tag{L7}$$

Proof: For the sake of contradiction, assume that $p@\{21\}$ is never established. Let $i = p.idx$ and $q = p.pos$. By (L5), $q@\{0\} \vee (q@\{6\} \wedge q.idx = i)$ is eventually established.

First, assume that $q@\{0\}$ is established. If q remains in its noncritical section forever, then by (I33), $Active[q] = \mathit{false}$ holds forever. Thus, p eventually establishes $p@\{21\}$ by executing statement 14, a contradiction.

On the other hand, if q enters its entry section again, then it establishes $q@\{1\}$. In this case, by (L6), q eventually establishes $q@\{6\} \wedge q.idx = i$.

It follows that $q@\{6\} \wedge q.idx = i$ is eventually established. By (I5), and using $p@\{14, 15\}$, it follows that $q@\{6\}$ remains true forever. But then, by (I34), p eventually establishes $p@\{21\}$ by executing statement 15, a contradiction. □

Finally, by (L3), (L4), and (L7), it follows that each **await** statement in Algorithm G-CC eventually terminates. Thus, Algorithm G-CC is starvation-free.

APPENDIX F

CORRECTNESS PROOF FOR ALGORITHM T IN SECTION 8.3

In this appendix, we formally prove that ALGORITHM T, presented in Section 8.3, satisfies the Exclusion and Starvation-freedom properties. Our proof makes use of an auxiliary array AccessCount. For each node i, AccessCount$[i]$ counts the number of *fetch-and-update* invocations on $Lock[i][0]$ since it was last reset. In Figure E.1, ALGORITHM T is shown with AccessCount added. In addition, in order to facilitate the proof, several statements are changed to equivalent (but more explicit) ones.

First, the syntax of the **for** loop at line 29b has been changed. Recall that $m = \Theta(\sqrt{\log N})$ is the degree of the arbitration tree, and that the root node has index 1. We leave it to the reader to verify that the children of node i consist precisely of nodes indexed from $((i-1)\cdot m+2)$ to $(i \cdot m+1)$ (inclusive).

Second, the *fetch-and-update* and *fetch-and-reset* statements at statements 11 and 39 have been changed in order to expose their low-level details (as defined on page 231). Variable $p.counter[l]$ is a private counter variable that is used when p accesses $Lock[\mathsf{Node}(p,l)][0]$. Of course, the *fetch-and-update* statements at statements 41 and 43 also use a separate set of counter variables, which are not explicitly shown in Figure E.1.

We now explain in detail how AccessCount is used.

Initially, $Lock[i][0]$ equals $\perp$, and hence AccessCount$[i]$ equals 0. Afterwards, each time *fetch-and-update* is invoked on $Lock[i][0]$, AccessCount$[i]$ is also incremented (statement 39). On the other hand, if *fetch-and-reset* is invoked on $Lock[i][0]$ (statement 11), then we have two cases to consider. First, if *fetch-and-reset* successfully resets $Lock[i][0]$ to $\perp$, then AccessCount$[i]$ is also reset to zero. Otherwise, the behavior of *fetch-and-reset* and any future *fetch-and-update* operations (if any) is undefined. In order to indicate this, AccessCount$[i]$ is changed to a special value $\star$. (Later, statement 14 manually resets both $Lock[i][0]$ and AccessCount$[i]$.)

```
shared auxiliary variable
    AccessCount: array[1..MAX_NODE] of $(0..\infty, \star)$ initially 0

process $p$ ::    /* $0 \le p < N$ */
while true do
0:        Noncritical Section;
1:        $Spin[p] := false$;
2:   2a:  $Winner[\mathsf{Node}(p, \texttt{MAX\_LEVEL})][0] := p$; /* automatically acquire its leaf node */
     2b:  $lev,\ break\_level := \texttt{MAX\_LEVEL} - 1,\ 0$;
          repeat
3:   3a:      $result := AcquireNode(lev)$;
     3b:      if $(result = \texttt{PRIMARY\_WINNER}) \lor (result = \texttt{SECONDARY\_WINNER})$ then
     3c:          $lev := lev - 1$
              else
     3d:          $break\_level := lev$
              fi
     3e:  until $(lev = 0) \lor (break\_level > 0)$;
     3f:  $side :=$ if $result = \texttt{PRIMARY\_WINNER}$ then 0
                elseif $result = \texttt{SECONDARY\_WINNER}$ then 1
                else 2;
4:        if $side = 2$ then await $Spin[p]$ fi;                /* wait until promoted */
5:        $\texttt{Entry}_3(side)$;
6:        Critical Section;
7:        $\texttt{Wait}()$;                                      /* wait at the barrier */
8:   8a:  $\texttt{Exit}_3(side)$;
     8b:  for $lev := break\_level + 1$ to $\texttt{MAX\_LEVEL} - 1$ do
              /* reopen each non-leaf node $p$ has acquired */
     8c:      $n := \mathsf{Node}(p, lev)$;
9:            if $Winner[n][0] = p$ then                          /* primary winner */
10:               $Winner[n][0] := \bot$;
11:               /* $(prev, new) := fetch\text{-}and\text{-}reset(Lock[n][0])$; */
                      $prev := Lock[n][0]$;
                      $new := \phi(prev, \beta_p[counter[lev]])$;    $Lock[n][0] := new$;
                  $\mathsf{AccessCount}[n] :=$ if $(new = \bot)$ then 0 else $\star$;
                  if $(n > 1) \land (prev \ne lock[lev])$ then
12:                   repeat $proc := Waiter[n]$ until $proc \ne \bot$;
13:                   $Enqueue(WaitingQueue, proc)$
                  fi;
14:               if $new \ne \bot$ then
                      $Lock[n][0] := \bot$;
                      $\mathsf{AccessCount}[n] := 0$ fi
15:           elseif $Winner[n][1] = p$ then                      /* secondary winner */
16:               $Winner[n][1] := \bot$;
17:               $Lock[n][1] := \bot$;
18:               if $WaiterLock[n] \ne \bot$ then
19:                   repeat $proc := Waiter[n]$ until $proc \ne \bot$;
20:                   $Enqueue(WaitingQueue, proc)$
              fi fi
          od;
```

Figure F.1: ALGORITHM T with auxiliary variables added. Non-auxiliary variables are as defined in Figure 8.9. (Continued on the next page.)

```
21:        $n$ := if $side = 2$ then Node($p$, $break\_level$)    /* promoted at node $n$ */
                else 1;    /* the root node */
22:        if $Waiter[n] = p$ then                                 /* primary waiter */
23:            $Waiter[n] := \bot$;
24:            $WaiterLock[n] := \bot$
           fi;
25:  25:   if $(n > 1) \wedge (Lock[n][0] \neq \bot)$ then
26:            repeat $proc := Winner[n][0]$ until $proc \neq \bot$;
27:            $Winner[n][0] := \bot$;
28:            $Lock[n][0] := \bot$;
               AccessCount[$n$] := 0;
29:  29a:      $Enqueue(WaitingQueue, proc)$
           fi;
     29b:  for $child := ((n-1) \cdot m + 2)$ to $(n \cdot m + 1)$ do
           /* equivalent to "for each $child$ := (a child of $n$) do" */
           /* $m$ = degree of the arbitration tree */
     29c:      for $i := 0$ to 1 do
30:                $proc := Winner[child][i]$;
31:                if $proc \neq \bot$ then $Enqueue(WaitingQueue, proc)$ fi
           od od;
32:        $Winner$[Node($p$, MAX_LEVEL)][0] := $\bot$;           /* reopen its leaf node */
33:        $Remove(WaitingQueue, p)$;
34:        $proc := Promoted$;
           if $(proc = p) \vee (proc = \bot)$ then
35:            $proc := Dequeue(WaitingQueue)$;
36:            $Promoted := proc$;
37:            if $proc \neq \bot$ then $Spin[proc] := true$ fi
           fi;
38:        Signal()                                               /* open the barrier */
      od
```

Figure F.1: Algorithm T with auxiliary variables added, continued. (Continued on the next page.)

As before, we assume that each labeled sequence of statement(s) is atomic. (Some statements span multiple lines; line numbers in the sans serif font are used only for the purpose of description.) To avoid possible confusion, we give here a detailed account of several statements that are rather complicated. If a process p executes statement 3, then p executes lines 3a and L1, and establishes $p@\{39\}$. Note that lines 3b–3f are not executed in this case. On the other hand, if p executes statement 40, then p executes lines L6, L7, and some of 3b–3f, and establishes either $p@\{3\}$ or $p@\{4\}$. Statements 42 and 44 are executed similarly.

As a last example, assume that p executes statement 43. If $Lock[p.n][1] = \bot$ holds before its execution, then p executes lines L12 and L13, and establishes $p@\{44\}$. (In this case, p becomes a secondary winner.) Otherwise, p executes lines L12, L13, L16,

```
procedure $AcquireNode(lev$: 1..MAX_LEVEL)
     L1:  $n := \mathsf{Node}(p, lev)$;
39:       /* $(prev, new) := fetch\text{-}and\text{-}update(Lock[n][0])$; */
     L2a:     $counter[lev] := counter[lev] + 1$;
     L2b:     $prev := Lock[n][0]$;
     L2c:     $new := \phi(prev, \alpha_p[counter[lev]])$;
     L2d:     $Lock[n][0] := new$;
     L3:  if AccessCount[$n$] $\neq \star$ then AccessCount[$n$] := AccessCount[$n$] + 1 fi;
     L4:  $lock[lev] := new$;
     L5:  if $prev = \bot$ then
40:  L6:      $Winner[n][0] := p$;
     L7:      return PRIMARY_WINNER
          else
41:  L8:      $(prev, new) := fetch\text{-}and\text{-}update(WaiterLock[n])$;
     L9:      if $prev = \bot$ then
42:  L10:         $Waiter[n] := p$;
     L11:         return PRIMARY_WAITER
              else
43:  L12:         $(prev, new) := fetch\text{-}and\text{-}update(Lock[n][1])$;
     L13:         if $prev = \bot$ then
44:  L14:             $Winner[n][1] := p$;
     L15:             return SECONDARY_WINNER
                  else
     L16:             return SECONDARY_WAITER
          fi fi fi
```

Figure F.1: Algorithm T with auxiliary variables added, continued.

3b, 3d, 3e, and 3f, and establishes $p@\{4\}$ (*i.e.*, p becomes a secondary waiter and exits the **repeat** loop).

The following definitions are used in the proof.

Definition: We define $\mathsf{lev}(i)$ to be the level of node i in the tree. In particular, node 1 (the root node) is at level 1 (*i.e.*, $\mathsf{lev}(1) = 1$), and each leaf node is at level MAX_LEVEL. □

Definition: Assume that $i = \mathsf{Node}(p, l)$ and $j = \mathsf{Node}(p, l + 1)$ holds for some level l. (That is, i is the node at level l that is visited by process p, and j is a child node of i (at level $l + 1$) that is visited by process p.) We define the condition $W(p, i, j, s)$ to be true if and only if $Winner[j][s] = p \ \wedge\ p@\{3, 4, 39..44\} \ \wedge\ D_k(p, i, j, s)$ holds for some k ($1 \leq k \leq 7$), where conditions D_1–D_7 are defined as follows. (For brevity, we sometimes use D_k to denote $D_k(p, i, j, s)$.)

$$D_1 \ = \ (\exists q :: q@\{25..29\} \ \wedge\ q.n = i),$$

$$D_2 \ = \ (\exists q :: q@\{30, 31\} \ \wedge\ q.n = i \ \wedge\ (q.child, q.i) < (j, s)),$$

$$\begin{aligned}
D_3 &= (\exists q :: q@\{30\} \ \wedge \ q.n = i \ \wedge \ (q.child, q.i) = (j, s)),\\
D_4 &= (\exists q :: q@\{31\} \ \wedge \ q.proc = p),\\
D_5 &= (p \in WaitingQueue),\\
D_6 &= (\exists q :: q@\{36, 37\} \ \wedge \ q.proc = p), \quad \text{and}\\
D_7 &= (Spin[p] = true).
\end{aligned}$$

□

Condition $W(p, i, j, s)$ is used to prove starvation-freedom. As shown later, if p is either the primary winner or the secondary winner of node j and is waiting at node i, then $W(p, i, j, s)$ is eventually established. Moreover, once $W(p, i, j, s)$ is established, D_7 is also eventually established (*i.e.*, p is promoted to its critical section).

F.1 List of Invariants

We will establish the Exclusion property by proving that the conjunction of a number of assertions is an invariant. This proves that each of these assertions individually is an invariant. These invariants are listed below. Unless stated otherwise, we assume the following: i and j range over the set of node indices; l ranges over 1..MAX_LEVEL; s ranges over 0..1; p, q, and r range over $0..N-1$.

invariant **(Exclusion)** $\left|\{p :: p@\{6..8\}\}\right| \leq 1$ (I1)

invariant $\left|\{p :: p@\{8..38\}\}\right| \leq 1$ (I2)

invariant $p@\{4..10\} \ \wedge \ p.side = 0 \ \Rightarrow \ Winner[1][0] = p$ (I3)

invariant $p@\{4..16\} \ \wedge \ p.side = 1 \ \Rightarrow \ Winner[1][1] = p$ (I4)

invariant $p@\{5..36\} \ \wedge \ p.side = 2 \ \Rightarrow \ Promoted = p$ (I5)

invariant $p@\{10\} \ \Rightarrow \ Winner[p.n][0] = p$ (I6)

invariant $p@\{16\} \ \Rightarrow \ Winner[p.n][1] = p$ (I7)

invariant $p@\{23\} \ \Rightarrow \ Waiter[p.n] = p$ (I8)

invariant $p@\{40\} \ \Rightarrow \ Lock[p.n][0] \neq \bot \ \wedge \ Winner[p.n][0] = \bot$ (I9)

invariant $p@\{44\} \ \Rightarrow \ Lock[p.n][1] \neq \bot \ \wedge \ Winner[p.n][1] = \bot$ (I10)

invariant $p@\{42\} \ \Rightarrow \ WaiterLock[p.n] \neq \bot \ \wedge \ Waiter[p.n] = \bot$ (I11)

invariant $p@\{2..4, 39..44\} \ \wedge \ Spin[p] = true \ \Rightarrow \ Promoted = p$ (I12)

invariant $Lock[i][s] = \bot \ \wedge \ \mathsf{lev}(i) < \texttt{MAX\_LEVEL} \ \Rightarrow \ Winner[i][s] = \bot$ (I13)

invariant $\mathit{WaiterLock}[i] = \bot \;\Rightarrow\; \mathit{Waiter}[i] = \bot$ (I14)

invariant $\left|\{p :: p@\{40\} \;\wedge\; p.n = i\}\right| \leq 1$ (I15)

invariant $\left|\{p :: p@\{44\} \;\wedge\; p.n = i\}\right| \leq 1$ (I16)

invariant $\left|\{p :: p@\{42\} \;\wedge\; p.n = i\}\right| \leq 1$ (I17)

invariant $p@\{11, 28\} \;\Rightarrow$
$\quad \mathit{Lock}[p.n][0] \neq \bot \;\wedge\; \mathit{Winner}[p.n][0] = \bot \;\wedge$
$\quad \neg(\exists q :: q@\{40\} \;\wedge\; q.n = p.n)$ (I18)

invariant $p@\{12..14\} \;\wedge\; p.\mathit{new} \neq \bot \;\Rightarrow$
$\quad \mathit{Lock}[p.n][0] \neq \bot \;\wedge\; \mathit{Winner}[p.n][0] = \bot \;\wedge\; \mathsf{AccessCount}[p.n] = \star \;\wedge$
$\quad \neg(\exists q :: q@\{40\} \;\wedge\; q.n = p.n)$ (I19)

invariant $p@\{17\} \;\Rightarrow$
$\quad \mathit{Lock}[p.n][1] \neq \bot \;\wedge\; \mathit{Winner}[p.n][1] = \bot \;\wedge$
$\quad \neg(\exists q :: q@\{44\} \;\wedge\; q.n = p.n)$ (I20)

invariant $p@\{24\} \;\Rightarrow$
$\quad \mathit{WaiterLock}[p.n] \neq \bot \;\wedge\; \mathit{Waiter}[p.n] = \bot \;\wedge$
$\quad \neg(\exists q :: q@\{42\} \;\wedge\; q.n = p.n)$ (I21)

invariant $p@\{27\} \;\Rightarrow\; \mathit{Winner}[p.n][0] = p.\mathit{proc} \neq \bot$ (I22)

invariant $p@\{40\} \;\wedge\; \mathit{Lock}[p.n][0] \neq p.\mathit{lock}[\mathsf{lev}(p.n)] \;\wedge\; p.n > 1 \;\Rightarrow$
$\quad (\exists q :: q@\{25, 26, 41\} \;\wedge\; q.n = p.n) \;\vee\; \mathit{WaiterLock}[p.n] \neq \bot$ (I23)

invariant $\mathit{Winner}[i][0] = p \;\wedge\; \mathit{Lock}[i][0] \neq p.\mathit{lock}[\mathsf{lev}(i)] \;\wedge$
$(1 < \mathsf{lev}(i) < \mathtt{MAX\_LEVEL}) \;\Rightarrow$
$\quad (\exists q :: q@\{25..27, 41\} \;\wedge\; q.n = i) \;\vee\; \mathit{WaiterLock}[i] \neq \bot$ (I24)

invariant $p@\{11\} \;\wedge\; \mathit{Lock}[p.n][0] \neq p.\mathit{lock}[\mathsf{lev}(p.n)] \;\wedge\; p.n > 1 \;\Rightarrow$
$\quad (\exists q :: q@\{41\} \;\wedge\; q.n = p.n) \;\vee\; \mathit{WaiterLock}[p.n] \neq \bot$ (I25)

invariant $p@\{12\} \;\Rightarrow\; (\exists q :: q@\{41\} \;\wedge\; q.n = p.n) \;\vee\; \mathit{WaiterLock}[p.n] \neq \bot$ (I26)

invariant $p@\{19\} \;\Rightarrow\; \mathit{WaiterLock}[p.n] \neq \bot$ (I27)

invariant $\mathit{WaiterLock}[i] \neq \bot \;\Rightarrow$
$\quad (\exists p :: p@\{24, 42\} \;\wedge\; p.n = i) \;\vee\; \mathit{Waiter}[i] \neq \bot$ (I28)

invariant $p@\{26\} \;\Rightarrow\; \mathit{Lock}[p.n][0] \neq \bot$ (I29)

invariant $\mathit{Lock}[i][0] \neq \bot \;\wedge\; \mathsf{lev}(i) < \mathtt{MAX\_LEVEL} \;\Rightarrow$
$\quad (\exists p :: p@\{11, 28, 40\} \;\wedge\; p.n = i) \;\vee$
$\quad \mathit{Winner}[i][0] \neq \bot \;\vee$
$\quad (\exists p :: p@\{12..14\} \;\wedge\; p.n = i \;\wedge\; p.\mathit{new} \neq \bot)$ (I30)

invariant $(\mathsf{AccessCount}[i] = 0) = (\mathit{Lock}[i][0] = \bot)$ (I31)

invariant $\mathit{Winner}[i][0] = p \;\wedge\; \mathsf{lev}(i) < \mathtt{MAX\_LEVEL} \;\Rightarrow$

$$(p@\{3, 39..44\} \ \wedge \ p.lev < \mathsf{lev}(i)) \ \vee$$
$$(p@\{4..8\} \ \wedge \ p.break\_level < \mathsf{lev}(i)) \ \vee$$
$$(p@\{9..20\} \ \wedge \ p.lev < \mathsf{lev}(i)) \ \vee$$
$$(p@\{9, 10\} \ \wedge \ p.lev = \mathsf{lev}(i)) \qquad \text{(I32)}$$

invariant $Winner[i][1] = p \ \Rightarrow$
$$(p@\{3, 39..44\} \ \wedge \ p.lev < \mathsf{lev}(i)) \ \vee$$
$$(p@\{4..8\} \ \wedge \ p.break\_level < \mathsf{lev}(i)) \ \vee$$
$$(p@\{9..20\} \ \wedge \ p.lev < \mathsf{lev}(i)) \ \vee$$
$$(p@\{9, 15, 16\} \ \wedge \ p.lev = \mathsf{lev}(i)) \qquad \text{(I33)}$$

invariant $Winner[i][1] = p \ \Rightarrow \ Winner[i][0] \neq p$ (I34)

invariant $Waiter[i] = p \ \Rightarrow$
$$p@\{4..23\} \ \wedge \ p.break\_level = \mathsf{lev}(i) \ \wedge \ p.side = 2 \ \wedge$$
$$(p@\{22, 23\} \ \Rightarrow \ p.n = i) \qquad \text{(I35)}$$

invariant $\mathsf{AccessCount}[i] = 1 \ \Rightarrow$
$$(\exists p :: p@\{11, 40\} \ \wedge \ p.n = i \ \wedge \ p.lock[\mathsf{lev}(i)] = Lock[i][0]) \ \vee$$
$$(\exists p :: \ Winner[i][0] = p \ \wedge \ p.lock[\mathsf{lev}(i)] = Lock[i][0]) \ \vee$$
$$(\exists q :: q@\{28\} \ \wedge \ q.n = i) \qquad \text{(I36)}$$

invariant $\mathsf{AccessCount}[i] = 2 \ \Rightarrow$
$$(\exists p :: p@\{11, 40\} \ \wedge \ p.n = i \ \wedge \ p.lock[\mathsf{lev}(i)] \neq Lock[i][0]) \ \vee$$
$$(\exists p :: \ Winner[i][0] = p \ \wedge \ p.lock[\mathsf{lev}(i)] \neq Lock[i][0]) \ \vee$$
$$(\exists q :: q@\{28\} \ \wedge \ q.n = i) \qquad \text{(I37)}$$

invariant $p@\{41\} \ \wedge \ p.n = i = \mathsf{Node}(p, l) \ \wedge \ j = \mathsf{Node}(p, l + 1) \ \wedge$
$Winner[j][s] = p \ \wedge \ WaiterLock[i] = \bot \ \wedge \ i > 1 \ \Rightarrow$
$$W(p, i, j, s) \ \vee \ \mathsf{AccessCount}[i] = 2 \ \vee$$
$$(\exists q : q \neq p :: q@\{12, 41\} \ \wedge \ q.n = i) \qquad \text{(I38)}$$

invariant $\mathsf{AccessCount}[i] \geq 2 \ \wedge \ WaiterLock[i] = \bot \ \wedge \ i > 1 \ \Rightarrow$
$$(\exists p :: p@\{25..28, 41\} \ \wedge \ p.n = i) \qquad \text{(I39)}$$

invariant $p@\{13\} \ \Rightarrow \ Waiter[p.n] \neq \bot$ (I40)

invariant $p@\{14\} \ \wedge \ p.n > 1 \ \wedge \ p.new \neq \bot \ \Rightarrow \ Waiter[p.n] \neq \bot$ (I41)

invariant $p@\{42\} \ \wedge \ p.n = i = \mathsf{Node}(p, l) \ \wedge \ j = \mathsf{Node}(p, l + 1) \ \wedge$
$Winner[j][s] = p \ \wedge \ i > 1 \ \Rightarrow$
$$W(p, i, j, s) \ \vee \ \mathsf{AccessCount}[i] = 2 \ \vee \ Lock[i][1] \neq \bot \ \vee$$
$$(\exists q : q \neq p :: q@\{12, 18, 19, 41, 43\} \ \wedge \ q.n = i) \qquad \text{(I42)}$$

invariant $p@\{4\} \ \wedge \ Waiter[i] = p \ \wedge$
$i = \mathsf{Node}(p, l) \ \wedge \ j = \mathsf{Node}(p, l + 1) \ \wedge$

$$\begin{aligned}
&\quad \mathit{Winner}[j][s] = p \ \wedge\ i > 1 \ \Rightarrow \\
&\qquad W(p,i,j,s) \ \vee\ \mathsf{AccessCount}[i] = 2 \ \vee\ \mathit{Lock}[i][1] \neq \bot \ \vee \\
&\qquad (\exists q : q \neq p :: q@\{12,18,19,41,43\} \ \wedge\ q.n = i) \ \vee \\
&\qquad (\exists q : q \neq p :: q@\{13,20\} \ \wedge\ q.\mathit{proc} = p) && \text{(I43)}
\end{aligned}$$

$$\begin{aligned}
\textbf{invariant}\ &\big[(p@\{43\} \ \wedge\ p.n = i) \ \vee \\
&\qquad (p@\{4\} \ \wedge\ p.\mathit{result} = \mathtt{SECONDARY\_WAITER} \ \wedge\ p.\mathit{break\_level} = l)\big] \ \wedge \\
&i = \mathsf{Node}(p,l) \ \wedge\ j = \mathsf{Node}(p,l+1) \ \wedge \\
&\mathit{Winner}[j][s] = p \ \wedge\ \mathit{WaiterLock}[i] = \bot \ \wedge\ i > 1 \ \Rightarrow \\
&\qquad\quad W(p,i,j,s) && \text{(I44)}
\end{aligned}$$

$$\begin{aligned}
\textbf{invariant}\ &\big[(p@\{41..43\} \ \wedge\ p.n = 1) \ \vee\ (p@\{4\} \ \wedge\ p.\mathit{break\_level} = 1)\big] \ \wedge \\
&j = \mathsf{Node}(p,2) \ \wedge\ \mathit{Winner}[j][s] = p \ \Rightarrow \\
&\qquad W(p,1,j,s) \ \vee\ \mathit{Lock}[1][0] \neq \bot \ \vee \\
&\qquad (\exists q :: q@\{9..25\} \ \wedge\ q.\mathit{break\_level} = 0) && \text{(I45)}
\end{aligned}$$

$$\textbf{invariant}\ \ p \in \mathit{WaitingQueue} \ \Rightarrow\ p@\{3..33, 39..44\} \qquad \text{(I46)}$$

$$\textbf{invariant}\ \ p@\{13,20,27..29,31,36\} \ \wedge\ p.\mathit{proc} = q \ \wedge\ p \neq q \ \Rightarrow\ q@\{3..7,39..44\} \qquad \text{(I47)}$$

$$\textbf{invariant}\ \ \mathit{Promoted} = p \ \Rightarrow\ p@\{3..36, 39..44\} \qquad \text{(I48)}$$

$$\textbf{invariant}\ \ p@\{36\} \ \Rightarrow\ p.\mathit{proc} \neq p \qquad \text{(I49)}$$

$$\begin{aligned}
\textbf{invariant}\ &\|\mathit{WaitingQueue}\| > 0 \ \Rightarrow \\
&\qquad (\exists p :: p@\{3,4,39..44\} \ \wedge\ \mathit{Spin}[p] = \mathit{true}) \ \vee \\
&\qquad (\exists p :: p@\{5..35\}) \ \vee \\
&\qquad (\exists p,q :: p@\{3,4,39..44\} \ \wedge\ q@\{36,37\} \ \wedge\ q.\mathit{proc} = p) && \text{(I50)}
\end{aligned}$$

$$\textbf{invariant}\ \ \mathit{Promoted} = p \ \wedge\ \mathit{Spin}[p] = \mathit{false} \ \Rightarrow\ (\exists q :: q@\{37\} \ \wedge\ q.\mathit{proc} = p) \qquad \text{(I51)}$$

$$\begin{aligned}
\textbf{invariant}\ &\big[(p@\{3,39..44\} \ \wedge\ p.\mathit{lev} < l) \ \vee\ (p@\{4\} \ \wedge\ p.\mathit{break\_level} < l)\big] \ \wedge \\
&i = \mathsf{Node}(p,l) \ \Rightarrow \\
&\qquad \mathit{Winner}[i][0] = p \ \vee\ \mathit{Winner}[i][1] = p \ \vee \\
&\qquad (\exists q :: q@\{28,29,36,37\} \ \wedge\ q.\mathit{proc} = p) \ \vee \\
&\qquad p \in \mathit{WaitingQueue} \ \vee \\
&\qquad \mathit{Spin}[p] = \mathit{true} && \text{(I52)}
\end{aligned}$$

$$\begin{aligned}
\textbf{invariant}\ &p@\{9..20, 39..44\} \ \Rightarrow \\
&\qquad p.n = \mathsf{Node}(p, p.\mathit{lev}) \ \wedge\ p.\mathit{lev} = \mathsf{lev}(p.n) < \mathtt{MAX\_LEVEL} && \text{(I53)}
\end{aligned}$$

$$\textbf{invariant}\ \ p@\{4..38\} \ \Rightarrow\ \big[(p.\mathit{break\_level} > 0) = (p.\mathit{side} = 2)\big] \qquad \text{(I54)}$$

$$\textbf{invariant}\ \ p@\{35,36\} \ \Rightarrow\ \mathit{Promoted} = p \ \vee\ \mathit{Promoted} = \bot \qquad \text{(I55)}$$

$$\textbf{invariant}\ \ p@\{37\} \ \Rightarrow\ p.\mathit{proc} = \mathit{Promoted} \qquad \text{(I56)}$$

$$\textbf{invariant}\ \ p@\{22..31\} \ \Rightarrow\ \mathsf{lev}(p.n) < \mathtt{MAX\_LEVEL} \qquad \text{(I57)}$$

$$\textbf{invariant}\ \ \mathit{Winner}[i][s] = p \ \vee\ \mathit{Waiter}[i] = p \ \Rightarrow\ i = \mathsf{Node}(p, \mathsf{lev}(i)) \qquad \text{(I58)}$$

invariant $\mathit{Winner}[i][0] = p \ \wedge\ \mathsf{lev}(i) = \texttt{MAX\_LEVEL} \ \Rightarrow\ p@\{3..32, 39..44\}$ (I59)

invariant $\mathsf{AccessCount}[i] = \star \ \Rightarrow\ (\exists p :: p@\{12..14\} \ \wedge\ p.n = i \ \wedge\ p.\mathit{new} \neq \bot)$ (I60)

invariant $p@\{22..31\} \ \wedge\ p.\mathit{break\_level} = 0 \ \Rightarrow\ p.n = 1$ (I61)

invariant $p@\{3, 39..44\} \ \Rightarrow\ p.\mathit{lev} > 0 \ \wedge\ p.\mathit{break\_level} = 0$ (I62)

invariant $\mathit{Winner}[i][1] \neq \bot \ \Rightarrow\ \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$ (I63)

invariant $(\mathit{Winner}[i][0] = p \ \wedge\ \mathsf{lev}(i) < \texttt{MAX\_LEVEL}) \ \vee$
$(p@\{11, 40\} \ \wedge\ p.n = i) \ \Rightarrow$
$p.\mathit{lock}[\mathsf{lev}(i)] = \phi(\bot, \alpha_p[p.\mathit{counter}[\mathsf{lev}(i)]])$ (I64)

invariant $\mathit{Lock}[i][1] \neq \bot \ \Rightarrow\ (\exists p :: p@\{17, 44\} \ \wedge\ p.n = i) \ \vee\ \mathit{Winner}[i][1] \neq \bot$ (I65)

invariant $p@\{4\} \ \wedge\ p.\mathit{result} = \texttt{PRIMARY\_WAITER} \ \Rightarrow$
$\mathit{Waiter}[\mathsf{Node}(p, p.\mathit{break\_level})] = p$ (I66)

F.2 Proof of the Exclusion Property

We now prove that each of (I1)–(I66) is an invariant. For each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.) If I is an implication (which is the case for most of our invariants), then it suffices to check only those program statements that may establish the antecedent of I, or that may falsify the consequent if executed while the antecedent holds.

To facilitate the proofs, we will index statements 9–20, 22–31, and 39–44 by the node being accessed (*i.e.*, the private variable n). For example, $40[i].p$ denotes the execution of statement 40 by p when its private variable $p.n$ equals i.

invariant (Exclusion) $\left|\{p :: p@\{6..8\}\}\right| \leq 1$ (I1)

Proof: Since the $\texttt{Entry}_3$ and $\texttt{Exit}_3$ routines (statements 5 and 8) are assumed to be correct, (I1) follows easily from (I3), (I4), and (I5). □

invariant $\left|\{p :: p@\{8..38\}\}\right| \leq 1$ (I2)

Proof: Since the `Wait` and `Signal` routines (statements 7 and 38) are assumed to be correct, (I2) follows easily from (I1). □

invariant $p@\{4..10\} \ \wedge\ p.\mathit{side} = 0 \ \Rightarrow\ \mathit{Winner}[1][0] = p$ (I3)

Proof: The only statements that may establish the antecedent are 40.p, 42.p, 43.p, and 44.p. (Note that statement 3.p establishes $p@\{39\}$.) However, the antecedent (in particular, $p.side = 0$) may be established only if line 3f is executed when $p.result =$ PRIMARY_WINNER holds. Thus, only statement 40.p may establish the antecedent.

In this case, p first executes lines L6, L7, 3b, 3c, and 3e. By (I62), $p.break\_level = 0$ holds throught the execution of these lines. Thus, p may establish $p@\{4\}$ (by exiting the **repeat** loop and executing line 3f) only if $p.lev = 0$ holds when line 3e is executed. Due to line 3c, it follows that statement 40.p may establish $p@\{4\}$ only if executed when $p.lev = 1$ holds.

By (I53), this also implies $p.n = \mathsf{Node}(p, 1) = 1$. Thus, statement 40.$p$ establishes the consequent in this case.

The only statements that may falsify the consequent are 2.q, 10[1].q, 27[1].q, 32.q, and 40[1].q, where q is any arbitrary process. Statements 2.q and 32.q access a leaf node, and hence cannot update $Winner[1][0]$.

By (I6), $Winner[1][0] = q$ holds before the execution of 10[1].q. Taken together with the consequent, we have $p = q$, and hence statement 10[1].q falsifies the antecedent.

Statement 27.q is executed only if $q.n > 1$. Thus, statement 27.q is never executed when $q.n = 1$ holds.

By (I9), the consequent is false before the execution of statement 40[1].q. Thus, statement 40[1].q cannot falsify the consequent. □

invariant $p@\{4..16\} \land p.side = 1 \Rightarrow Winner[1][1] = p$ (I4)

Proof: The only statements that may establish the antecedent are 40.p, 42.p, 43.p, and 44.p. However, the antecedent (in particular, $p.side = 1$) may be established only if line 3f is executed when $p.result =$ SECONDARY_WINNER holds. Thus, only statement 44.p may establish the antecedent. By an argument similar to that in the proof of (I3), it follows that statement 44.p may establish the antecedent only if executed when $p.lev = 1$ holds. By (I53), this also implies $p.n = \mathsf{Node}(p, 1) = 1$. Thus, statement 44.$p$ establishes the consequent in this case.

The only statements that may falsify the consequent are 16[1].q and 44[1].q, where q is any arbitrary process.

By (I7), $Winner[1][1] = q$ holds before the execution of 16[1].q. Taken together with the consequent, we have $p = q$, and hence statement 16[1].q falsifies the antecedent.

By (I10), the consequent is false before the execution of statement $44[1].q$. Thus, statement $44[1].q$ cannot falsify the consequent. □

invariant $p@\{5..36\} \wedge p.side = 2 \Rightarrow Promoted = p$ (I5)

Proof: The only statement that may establish the antecedent is $4.p$, which may do so only if $Spin[p] = true$ holds. By (I12), this in turn implies the consequent.

The only statement that may falsify the consequent is $36.q$. By (I55), $Promoted = q \vee Promoted = \perp$ holds before its execution. Taken together with the consequent, we have $p = q$, and hence statement $36.q$ falsifies the antecedent. □

invariant $p@\{10\} \Rightarrow Winner[p.n][0] = p$ (I6)

Proof: The only statement that may establish the antecedent is $9.p$, which may do so only if the consequent is true.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $2.q$, $10[i].q$, $27[i].q$, $32.q$, and $40[i].q$, where $i = p.n$ and q is any arbitrary process. By (I53), the antecedent implies $\mathsf{lev}(p.n) < \texttt{MAX\_LEVEL}$. Since statements $2.q$ and $32.q$ access a leaf node, they cannot update $Winner[p.n][0]$ while $p@\{10\}$ holds.

By (I2), if statement $10[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $10[i].q$ falsifies the antecedent. Also, by (I2), statement $27[i].q$ cannot be executed while the antecedent holds.

By (I9), the consequent is false before the execution of statement $40[i].q$. Thus, statement $40[i].q$ cannot falsify the consequent. □

invariant $p@\{16\} \Rightarrow Winner[p.n][1] = p$ (I7)

Proof: The only statement that may establish the antecedent is $15.p$, which may do so only if the consequent is true.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $16[i].q$ and $44[i].q$, where $i = p.n$ and q is any arbitrary process. By (I2), if statement $16[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $16[i].q$ falsifies the antecedent.

By (I10), the consequent is false before the execution of statement $44[i].q$. Thus, statement $44[i].q$ cannot falsify the consequent. □

invariant $p@\{23\} \;\Rightarrow\; \mathit{Waiter}[p.n] = p$ (I8)

Proof: The only statement that may establish the antecedent is $22.p$, which may do so only if the consequent is true.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $23[i].q$ and $42[i].q$, where $i = p.n$ and q is any arbitrary process. By (I2), if statement $23[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $23[i].q$ falsifies the antecedent.

By (I11), the consequent is false before the execution of statement $42[i].q$. Thus, statement $42[i].q$ cannot falsify the consequent. □

invariant $p@\{40\} \;\Rightarrow\; \mathit{Lock}[p.n][0] \neq \bot \;\wedge\; \mathit{Winner}[p.n][0] = \bot$ (I9)

Proof: The only statement that may establish the antecedent is $39.p$, which may do so only if executed when $\mathit{Lock}[p.n][0] = \bot$ holds. In this case, by the semantics of *fetch-and-update*, statement $39.p$ establishes $\mathit{Lock}[p.n][0] \neq \bot$. Also, By (I53), $p@\{39\}$ implies $\mathsf{lev}(p.n) < \texttt{MAX\_LEVEL}$, and hence, by applying (I13) with 'i' $\leftarrow p.n$ and 's' $\leftarrow 0$, $\mathit{Winner}[p.n][0] = \bot$ holds before and after the execution of statement $39.p$.

Let $i = p.n$. Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $11[i].q$, $14[i].q$, and $28[i].q$ (which may establish $\mathit{Lock}[p.n][0] = \bot$) and $2.q$ and $40[i].q$ (which may establish $\mathit{Winner}[p.n][0] \neq \bot$), where q is any arbitrary process. (Note that statement $39[i].q$ cannot establish $\mathit{Lock}[p.n][0] = \bot$, since we assume that the underlying *fetch-and-ϕ* primitive is self-resettable.) By applying (I18) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, it follows that statements $11[i].q$ and $28[i].q$ cannot be executed while the antecedent holds. Statement $14[i].q$ updates $\mathit{Lock}[p.n][0]$ only if $q.\mathit{new} \neq \bot$ holds, which contradicts the antecedent by (I19) (with 'p' $\leftarrow q$ and 'q' $\leftarrow p$).

By (I53), the antecedent implies $\mathsf{lev}(i) < \texttt{MAX\_LEVEL}$. Since statement $2.q$ accesses a leaf node, it cannot update $\mathit{Winner}[i][0]$ while the antecedent holds.

By (I15), if statement $40[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $40[i].q$ falsifies the antecedent. □

invariant $p@\{44\} \;\Rightarrow\; \mathit{Lock}[p.n][1] \neq \bot \;\wedge\; \mathit{Winner}[p.n][1] = \bot$ (I10)

Proof: The only statement that may establish the antecedent is $43.p$, which may do so only if executed when $\mathit{Lock}[p.n][1] = \bot$ holds. In this case, by the semantics of *fetch-and-update*, statement $43.p$ establishes $\mathit{Lock}[p.n][1] \neq \bot$. Also, By (I53), $p@\{43\}$ implies $\mathsf{lev}(p, n) < \texttt{MAX\_LEVEL}$, and hence, by applying (I13) with 'i' $\leftarrow p.n$ and 's' $\leftarrow 1$, $\mathit{Winner}[p.n][1] = \bot$ holds before and after the execution of statement $43.p$.

Let $i = p.n$. Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $17[i].q$ (which may establish $\mathit{Lock}[p.n][1] = \bot$) and $44[i].q$ (which may establish $\mathit{Winner}[p.n][1] \neq \bot$), where q is any arbitrary process. By applying (I20) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, it follows that statement $17[i].q$ cannot be executed while the antecedent holds. By (I16), if statement $44[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $44[i].q$ falsifies the antecedent. □

invariant $p@\{42\} \;\Rightarrow\; \mathit{WaiterLock}[p.n] \neq \bot \;\wedge\; \mathit{Waiter}[p.n] = \bot$ (I11)

Proof: The only statement that may establish the antecedent is $41.p$, which may do so only if executed when $\mathit{WaiterLock}[p.n] = \bot$ holds. In this case, by the semantics of *fetch-and-update*, statement $41.p$ establishes $\mathit{WaiterLock}[p.n] \neq \bot$. Also, by (I14), $\mathit{Waiter}[p.n] = \bot$ holds before and after the execution of statement $41.p$.

Let $i = p.n$. Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $24[i].q$ (which may establish $\mathit{WaiterLock}[p.n] = \bot$) and $42[i].q$ (which may establish $\mathit{Waiter}[p.n] \neq \bot$), where q is any arbitrary process. By applying (I21) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, it follows that statement $24[i].q$ cannot be executed while the antecedent holds. By (I17), if statement $42[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $42[i].q$ falsifies the antecedent. □

invariant $p@\{2..4, 39..44\} \;\wedge\; \mathit{Spin}[p] = \mathit{true} \;\Rightarrow\; \mathit{Promoted} = p$ (I12)

Proof: The only statements that may establish the antecedent are $1.p$ and $37.q$, where q is any arbitrary process. However, statement $1.p$ assigns $\mathit{Spin}[p] := \mathit{false}$, and hence cannot establish the antecedent. Statement $37.q$ may establish the antecedent only if $q.\mathit{proc} = p$ holds, in which case, by (I56), the consequent holds as well.

The only statement that may falsify the consequent is $36.q$, where q is any arbitrary process. If $36.q$ is executed when the antecedent holds, then $q \neq p$. In this case, by

(I55), the consequent is false before the execution of $36.q$. It follows that statement $36.q$ cannot falsify the consequent. $\Box$

invariant $Lock[i][s] = \bot \ \wedge\ \mathsf{lev}(i) < \texttt{MAX\_LEVEL} \ \Rightarrow\ Winner[i][s] = \bot$ (I13)

Proof: The only statements that may establish the antecedent are $11[i].p$, $14[i].p$, and $28[i].p$ (for $s = 0$) and $17[i].p$ (for $s = 1$), where p is any arbitrary process. The consequent is true before and after the execution of each of $11[i].p$, $17[i].p$, and $28[i].p$, by invariants (I18), (I20), and (I18), respectively.

Statement $14[i].p$ may establish the antecedent only if $p.new \neq \bot$ holds, which implies the consequent by (I19).

The only statements that may falsify the consequent are $40[i].p$ (for $s = 0$) and and $44[i].p$ (for $s = 1$), where p is any arbitrary process. By (I9) and (I10), these statements cannot be executed while the antecedent holds. $\Box$

invariant $WaiterLock[i] = \bot \ \Rightarrow\ Waiter[i] = \bot$ (I14)

Proof: The only statement that may establish the antecedent is $24[i].p$, where p is any arbitrary process. By (I21), the antecedent is true before and after its execution.

The only statement that may falsify the consequent is $42[i].p$, where p is any arbitrary process. However, by (I11), statement $42[i].p$ cannot be executed while the antecedent holds. $\Box$

invariant $\left|\{p :: p@\{40\} \ \wedge\ p.n = i\}\right| \leq 1$ (I15)
invariant $\left|\{p :: p@\{44\} \ \wedge\ p.n = i\}\right| \leq 1$ (I16)
invariant $\left|\{p :: p@\{42\} \ \wedge\ p.n = i\}\right| \leq 1$ (I17)

Proof: Invariant (I15) might be potentially falsified only if a process p executes statement $39[i]$ while $q@\{40\} \wedge q.n = i$ holds for some process $q \neq p$. However, by applying (I9) with 'p' $\leftarrow q$, this implies that $Lock[i][0] \neq \bot$, and hence statement $39[i].p$ cannot establish $p@\{40\}$.

The proofs for invariants (I16) and (I17) are similar, except that (I10) and (I11) are used instead of (I9). $\Box$

invariant $p@\{11, 28\} \Rightarrow$

$$Lock[p.n][0] \neq \bot \ \wedge \ Winner[p.n][0] = \bot \ \wedge$$
$$\neg(\exists q :: q@\{40\} \ \wedge \ q.n = p.n) \qquad \text{(I18)}$$

Proof: Let $i = p.n$. The only statements that may establish the antecedent are $10.p$ and $27.p$. Statement $10.p$ establishes $Winner[i][0] = \bot$. Moreover, by (I6) and (I53), $Winner[i][0] = p \neq \bot \wedge \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$ holds before the execution of $10.p$. Therefore, by (I13) (with 's' $\leftarrow 0$), $Lock[i][0] \neq \bot$ holds before and after its execution. Also, by (I9), and using $Winner[i][0] \neq \bot$, it follows that $\neg(\exists q :: q@\{40\} \ \wedge \ q.n = i)$ also holds before, and hence after, the execution of $10.p$. It follows that the consequent holds after its execution.

Statement $27.p$ establishes $Winner[i][0] = \bot$. Moreover, by (I22) and (I57), $Winner[i][0] = p.proc \neq \bot \ \wedge \ \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$ holds before the execution of $27.p$. Therefore, by (I13) (with 's' $\leftarrow 0$), $Lock[i][0] \neq \bot$ holds before and after its execution. Also, by (I9), and using $Winner[i][0] \neq \bot$, it follows that $\neg(\exists q :: q@\{40\} \ \wedge \ q.n = i)$ also holds before, and hence after, the execution of $27.p$. It follows that the consequent holds after its execution.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $11[i].q$, $14[i].q$, and $28[i].q$ (which may establish $Lock[i][0] = \bot$), $40[i].q$ (which may establish $Winner[i][0] \neq \bot$), and $39[i].q$ (which may establish $q@\{40\} \ \wedge \ q.n = i$), where q is any arbitrary process. By (I2), if statements $11[i].q$ and $28[i].q$ are executed while the antecedent holds, then we have $p = q$. Thus, these statements falsify the antecedent. Also, by (I2), statement $14[i].q$ cannot be executed while the antecedent holds.

Statement $40[i].q$ cannot be executed while the consequent holds. Statement $39[i].q$ may falsify the consequent only if executed when $Lock[i][0] = \bot$ holds, which implies that the consequent is already false before its execution. □

invariant $p@\{12..14\} \ \wedge \ p.new \neq \bot \ \Rightarrow$

$$Lock[p.n][0] \neq \bot \ \wedge \ Winner[p.n][0] = \bot \ \wedge \ \mathsf{AccessCount}[p.n] = \star \ \wedge$$
$$\neg(\exists q :: q@\{40\} \ \wedge \ q.n = p.n) \qquad \text{(I19)}$$

Proof: Let $i = p.n$. The only statement that may establish the antecedent is $11.p$. Note that the consequent of (I18) is true before the execution of $11.p$. Moreover, if statement $11.p$ establishes $p.new \neq \bot$, then $\mathsf{AccessCount}[i] = \star \ \wedge \ Lock[i][0] \neq \bot$ holds after its execution. Thus, statement $11.p$ establishes the consequent.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $11[i].q$, $14[i].q$, and $28[i].q$ (which may update $Lock[i][0]$ and $\mathsf{AccessCount}[i]$), $40[i].q$ (which may establish $Winner[i][0] \neq \bot$), and $39[i].q$ (which may establish $q@\{40\} \wedge q.n = i$ and also update $Lock[i][0]$ and $\mathsf{AccessCount}[i]$), where q is any arbitrary process. By (I2), if statements $11[i].q$, $14[i].q$, and $28[i].q$ are executed while the antecedent holds, then we have $p = q$. As shown above, statement $11.p$ preserves (I19). Also, the antecedent is false after the execution of either $14.p$ or $28.p$.

Statement $40[i].q$ cannot be executed while the consequent holds. Statement $39[i].q$ cannot falsify $Lock[i][0] \neq \bot$ or $\mathsf{AccessCount}[i] = \star$. Moreover, it may establish $q@\{40\}$ only if executed when $Lock[i][0] = \bot$ holds, which implies that the consequent is already false before its execution. □

invariant $p@\{17\} \Rightarrow$

$$Lock[p.n][1] \neq \bot \ \wedge \ Winner[p.n][1] = \bot \ \wedge$$
$$\neg(\exists q :: q@\{44\} \ \wedge \ q.n = p.n) \qquad \text{(I20)}$$

Proof: Let $i = p.n$. The only statement that may establish the antecedent is $16.p$, which establishes $Winner[i][1] = \bot$. Moreover, by (I53) and (I7), $\mathsf{lev}(i) < \mathtt{MAX\_LEVEL} \wedge Winner[i][1] = p \neq \bot$ holds before its execution. Therefore, by (I13) (with 's' $\leftarrow$ 1), $Lock[i][1] \neq \bot$ holds before and after its execution. Moreover, by (I10), $\neg(\exists q :: q@\{44\} \wedge q.n = i)$ also holds before and after its execution. It follows that the consequent holds after the execution of $16.p$.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $17[i].q$ (which may establish $Lock[i][1] = \bot$), $44[i].q$ (which may establish $Winner[i][1] \neq \bot$), and $43[i].q$ (which may establish $q@\{44\} \wedge q.n = i$), where q is any arbitrary process. By (I2), if statement $17[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $17[i].q$ falsifies the antecedent.

Statement $44[i].q$ cannot be executed while the consequent holds. Statement $43[i].q$ may falsify the consequent only if executed when $Lock[i][1] = \bot$ holds, which implies that the consequent is already false before its execution. □

invariant $p@\{24\} \Rightarrow$

$$WaiterLock[p.n] \neq \bot \ \wedge \ Waiter[p.n] = \bot \ \wedge$$
$$\neg(\exists q :: q@\{42\} \ \wedge \ q.n = p.n) \qquad \text{(I21)}$$

Proof: The proof of (I21) is very similar to that of (I20). Let $i = p.n$. The only statement that may establish the antecedent is $23.p$, which establishes $\mathit{Waiter}[i] = \perp$. Moreover, by (I8), $\mathit{Waiter}[i] = p \neq \perp$ holds before its execution. Therefore, by (I14), $\mathit{WaiterLock}[i] \neq \perp$ holds before and after its execution. Moreover, by (I11), $\neg(\exists q :: q@\{42\} \ \wedge \ q.n = i)$ also holds before and after its execution. It follows that the consequent holds after the execution of $23.p$.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $24[i].q$ (which may establish $\mathit{WaiterLock}[i] = \perp$), $42[i].q$ (which may establish $\mathit{Waiter}[i] \neq \perp$), and $41[i].q$ (which may establish $q@\{42\} \ \wedge \ q.n = i$), where q is any arbitrary process. By (I2), if statement $24[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $24[i].q$ falsifies the antecedent.

Statement $42[i].q$ cannot be executed while the consequent holds. Statement $41[i].q$ may falsify the consequent only if executed when $\mathit{WaiterLock}[i] = \perp$ holds, which implies that the consequent is already false before its execution. $\square$

invariant $p@\{27\} \ \Rightarrow \ \mathit{Winner}[p.n][0] = p.\mathit{proc} \neq \perp$ (I22)

Proof: The only statement that may establish the antecedent is $26.p$; the consequent clearly holds after its execution.

Let $i = p.n$. Note that $p.n$ and $p.\mathit{proc}$ do not change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $2.q$, $10[i].q$, $27[i].q$, $32.q$, and $40[i].q$, where q is any arbitrary process. By (I57), the antecedent implies $\mathsf{lev}(i) < \texttt{MAX\_LEVEL}$. Since statements $2.q$ and $32.q$ access a leaf node, they cannot update $\mathit{Winner}[i][0]$.

By (I2), statement $10[i].q$ cannot be executed while the antecedent holds. Also, by (I2), if statement $27[i].q$ is executed while the antecedent holds, then we have $p = q$. Thus, statement $27[i].q$ falsifies the antecedent.

By (I9), the consequent is false before the execution of statement $40[i].q$. Thus, statement $40[i].q$ cannot falsify the consequent. $\square$

invariant $p@\{40\} \ \wedge \ \mathit{Lock}[p.n][0] \neq p.\mathit{lock}[\mathsf{lev}(p.n)] \ \wedge \ p.n > 1 \ \Rightarrow$
$(\exists q :: q@\{25, 26, 41\} \ \wedge \ q.n = p.n) \ \vee \ \mathit{WaiterLock}[p.n] \neq \perp$ (I23)

Proof: Let $i = p.n$. The only statements that may establish the antecedent are 39.p (which may establish $p@\{40\}$ and also update $p.lock[\mathsf{lev}(p.n)]$), 3.$p$, 8.$p$, 14.$p$, 15.$p$, 18.$p$, 20.$p$, and 21.$p$ (which may update $p.n$), and $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $Lock[i][0]$), where q is any arbitrary process, different from p. Statement 39.p establishes $Lock[i][0] = p.lock[\mathsf{lev}(i)]$. Hence, the antecedent is false after its execution.

The antecedent is false after the execution of each of 3.p, 8.p, 14.p, 15.p, 18.p, 20.p, and 21.p.

By applying (I18) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, it follows that statements $11[i].q$ and $28[i].q$ cannot be executed while $p@\{40\}$ holds. Statement $14[i].q$ may update $Lock[i][0]$ only if executed when $q.new \neq \perp$ holds. However, this contradicts $p@\{40\}$ by (I19) (with 'p' $\leftarrow q$ and 'q' $\leftarrow p$).

Statement $39[i].q$ may establish the antecedent only if executed when $p@\{40\}$ holds. By (I9), this implies that $Lock[i][0] \neq \perp$. Thus, statement $39[i].q$ establishes $q@\{41\} \wedge q.n = i$ in this case, which in turn implies the consequent.

Note that $p.n$ cannot change while the antecedent holds. Similarly, $q.n$ cannot change while $q@\{25, 26, 41\}$ holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $25[i].q$, $26[i].q$, and $41[i].q$ (which may falsify $q@\{25, 26, 41\}$), and $24[i].q$ (which establishes $WaiterLock[i] = \perp$), where q is any arbitrary process. Statement $24[i].q$ establishes $q@\{25\} \wedge q.n = i$.

If statement $25[i].q$ is executed while the antecedent holds, then by (I9), we have $Lock[i][0] \neq \perp$, and hence statement $25[i].q$ establishes $q@\{26\}$.

Statement $26[i].q$ may falsify $q@\{26\}$ only if executed when $Winner[i][0] \neq \perp$. By (I9), this implies that the antecedent is false.

Finally, by the semantics of *fetch-and-update*, after the execution of statement $41[i].q$, we have $WaiterLock[i] \neq \perp$, which implies the consequent. □

invariant $Winner[i][0] = p \wedge Lock[i][0] \neq p.lock[\mathsf{lev}(i)] \wedge$
$(1 < \mathsf{lev}(i) < \mathtt{MAX\_LEVEL}) \Rightarrow$
$(\exists q :: q@\{25..27, 41\} \wedge q.n = i) \vee WaiterLock[i] \neq \perp$ (I24)

Proof: The only statements that may establish the antecedent are 2.p and $40[i].p$ (which may establish $Winner[i][0] = p$), $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $\mathit{Lock}[i][0]$), and $39.p$ (which may update $p.\mathit{lock}[\mathsf{lev}(i)]$), where q is any arbitrary process. However, since $\mathsf{lev}(i) < \mathtt{MAX\_LEVEL}$, statement $2.p$ cannot establish the antecedent.

Statement $40[i].p$ may establish the antecedent only if $\mathit{Lock}[i][0] \neq p.\mathit{lock}[\mathsf{lev}(i)]$ holds, in which case the antecedent of (I23) holds. (Note that $\mathsf{lev}(i) > 1$ implies $i > 1$.) Thus, the consequent is true before and after the execution of $40[i].p$.

By (I18), statements $11[i].q$ and $28[i].q$ cannot be executed while $\mathit{Winner}[i][0] = p \neq \bot$ holds. Statement $14[i].q$ may update $\mathit{Lock}[i][0]$ only if executed when $q.\mathit{new} \neq \bot$ holds. However, this implies that $\mathit{Winner}[i][0] = \bot \neq p$ by (I19).

Statement $39[i].q$ may establish the antecedent only if executed when $\mathit{Winner}[i][0] = p \neq \bot$ holds. Since $\mathsf{lev}(i) < \mathtt{MAX\_LEVEL}$, by (I13) (with '$s$' $\leftarrow 0$), we have $\mathit{Lock}[i][0] \neq \bot$. Thus, statement $39[i].q$ establishes $q@\{41\} \wedge q.n = i$ in this case, which in turn implies the consequent.

Statement $39.p$ may potentially establish the antecedent by updating $p.\mathit{lock}[\mathsf{lev}(i)]$ only if executed when $\mathit{Winner}[i][0] = p \wedge p.\mathit{lev} = \mathsf{lev}(i) \wedge \mathsf{lev}(i) < \mathtt{MAX\_LEVEL}$ holds. However, by (I32), this and $p@\{39\}$ cannot hold simultaneously.

Note that $q.n$ cannot change while $q@\{25..27, 41\}$ holds. Thus, the only statements that may falsify the consequent are $25[i].q$, $27[i].q$, and $41[i].q$ (which may falsify $q@\{25..27, 41\}$), and $24[i].q$ (which establishes $\mathit{WaiterLock}[i] = \bot$), where q is any arbitrary process. Statement $24[i].q$ establishes $q@\{25\} \wedge q.n = i$.

If statement $25[i].q$ is executed while the antecedent holds, then by (I13) (with 's' $\leftarrow 0$), we have $\mathit{Lock}[i][0] \neq \bot$, and hence the statement establishes $q@\{26\}$. Statement $27[i].q$ falsifies the antecedent.

Finally, by the semantics of *fetch-and-update*, after the execution of statement $41[i].q$, we have $\mathit{WaiterLock}[i] \neq \bot$, which implies the consequent. $\square$

invariant $p@\{11\} \wedge \mathit{Lock}[p.n][0] \neq p.\mathit{lock}[\mathsf{lev}(p.n)] \wedge p.n > 1 \Rightarrow$
$$(\exists q :: q@\{41\} \wedge q.n = p.n) \vee \mathit{WaiterLock}[p.n] \neq \bot \qquad \text{(I25)}$$

Proof: Let $i = p.n$. The only statements that may establish the antecedent are $10.p$ (which establishes $p@\{11\}$), $3.p$, $8.p$, $14.p$, $15.p$, $18.p$, $20.p$, and $21.p$ (which may update $p.n$), and $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $\mathit{Lock}[i][0]$, and also $p.\mathit{lock}$, if $q = p$), where q is any arbitrary process. Statement $10.p$ may establish the antecedent only if executed when $\mathit{Lock}[i][0] \neq p.\mathit{lock}[\mathsf{lev}(i)]$ holds. Moreover, by (I6),

$\mathit{Winner}[i][0] = p$ holds before the execution of $10.p$. By (I53), we also have $\mathsf{lev}(i) < \mathtt{MAX\_LEVEL}$, and hence, by (I24), the following holds before the execution of $10.p$:

$$(\exists q :: q@\{25..27, 41\} \ \land \ q.n = i) \ \lor \ \mathit{WaiterLock}[i] \neq \bot.$$

By (I2), $p@\{11\}$ precludes $(\exists q :: q@\{25..27\})$, and hence we have the consequent.

The antecedent is false after the execution of each of $3.p$, $8.p$, $14.p$, $15.p$, $18.p$, $20.p$, and $21.p$.

If $p = q$, then the antecedent is false after the execution of each of statements $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$. Thus, assume $q \neq p$.

Statements $11[i].q$, $14[i].q$, and $28[i].q$ may establish the antecedent only if executed when $p@\{11\}$ holds, which is precluded by (I2).

Statement $39[i].q$ may establish the antecedent only if executed when $p@\{11\}$ holds. By (I18), this implies that $\mathit{Lock}[i][0] \neq \bot$. Thus, statement $39[i].q$ establishes $q@\{41\} \land q.n = i$ in this case, which in turn implies the consequent.

Note that $p.n$ cannot change while the antecedent holds. Similarly, $q.n$ cannot change while $q@\{41\}$ holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $41[i].q$ and $24[i].q$, where q is any arbitrary process. By (I2), statement $24[i].q$ cannot be executed while the antecedent holds. After the execution of $41[i].q$, we have $\mathit{WaiterLock}[i] \neq \bot$, which implies the consequent. $\square$

invariant $p@\{12\} \ \Rightarrow \ (\exists q :: q@\{41\} \ \land \ q.n = p.n) \ \lor \ \mathit{WaiterLock}[p.n] \neq \bot$ (I26)

Proof: Let $i = p.n$. The only statement that may establish the antecedent is $11.p$, which may do so only if executed when $\mathit{Lock}[i][0] \neq p.\mathit{lock}[p.\mathit{lev}] \ \land \ i > 1$ holds. By (I53), this is equivalent to $\mathit{Lock}[i][0] \neq p.\mathit{lock}[\mathsf{lev}(i)] \ \land \ i > 1$. Thus, by (I25), the consequent is true before and after the execution of $11.p$.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statements that may falsify the consequent while the antecedent holds are $41[i].q$ and $24[i].q$, where q is any arbitrary process. By (I2), statement $24[i].q$ cannot be executed while the antecedent holds. After the execution of $41[i].q$, we have $\mathit{WaiterLock}[i] \neq \bot$, which implies the consequent. $\square$

invariant $p@\{19\} \ \Rightarrow \ \mathit{WaiterLock}[p.n] \neq \bot$ (I27)

Proof: The only statement that may establish the antecedent is 18.p, which may do so only if the consequent holds. The only statement that may falsify the consequent is 24.q, where q is any arbitrary process. However, by (I2), statement 24.q cannot be executed while the antecedent holds. □

$$\textbf{invariant}\ \ \mathit{WaiterLock}[i] \neq \bot \ \Rightarrow \\ (\exists p :: p@\{24, 42\} \ \wedge \ p.n = i) \ \vee \ \mathit{Waiter}[i] \neq \bot \qquad \text{(I28)}$$

Proof: The only statement that may establish the antecedent is 41[i].p, where p is any arbitrary process. However, if statement 41[i].p is executed when the antecedent is false, then it establishes $p@\{42\} \ \wedge \ p.n = i$, which implies the consequent.

The only statements that may falsify the consequent are 23[i].p, 24[i].p, and 42[i].p. Statement 23[i].p establishes $p@\{24\} \ \wedge \ p.n = i$. Statement 24[$i$].$p$ falsifies the antecedent. Statement 42[i].p establishes $\mathit{Waiter}[i] \neq \bot$. □

$$\textbf{invariant}\ \ p@\{26\} \ \Rightarrow \ \mathit{Lock}[p.n][0] \neq \bot \qquad \text{(I29)}$$

Proof: The only statement that may establish the antecedent is 25.p, which may do so only if the consequent holds. The only statements that may falsify the consequent are 11[i].q, 14[i].q, 28[i].q, and 39[i].q, where q is any arbitrary process. By (I2), statements 11[i].q, 14[i].q, and 28[i].q cannot be executed while the antecedent holds. By the semantics of *fetch-and-update*, statement 39[i].q cannot falsify the consequent. □

$$\textbf{invariant}\ \ \mathit{Lock}[i][0] \neq \bot \ \wedge \ \mathsf{lev}(i) < \texttt{MAX\_LEVEL} \ \Rightarrow \\ (\exists p :: p@\{11, 28, 40\} \ \wedge \ p.n = i) \ \vee \\ \mathit{Winner}[i][0] \neq \bot \ \vee \\ (\exists p :: p@\{12..14\} \ \wedge \ p.n = i \ \wedge \ p.\mathit{new} \neq \bot) \qquad \text{(I30)}$$

Proof: The only statements that may establish the antecedent are 11[i].p and 39[i].p. By (I18), the antecedent is already true before the execution of statement 11[i].p. If statement 39[i].p is executed while the antecedent is false (*i.e.*, $\mathit{Lock}[i][0] = \bot$), then it establishes $p@\{40\} \ \wedge \ p.n = i$, which implies the consequent.

The only statements that may falsify the first disjunct of the consequent are 11[i].p, 28[i].p, and 40[i].p. If the antecedent holds after the execution of 11[i].p, then $p@\{12, 14\} \ \wedge \ p.n = i \ \wedge \ p.\mathit{new} \neq \bot$ also holds after its execution, which implies the third disjunct. The antecedent is false after the execution of $28[i].p$. After the execution of $40[i].p$, $Winner[i][0] \neq \perp$ holds.

The only statements that may falsify $Winner[i][0] \neq \perp$ are $10[i].p$ and $27[i].p$. Both establish the first disjunct.

The only statement that may falsify the third disjunct is $14.p$, where p is a process satisfying $p.n = i \wedge p.new \neq \perp$. In this case, the antecedent is false after the execution of $14.p$. □

invariant $(\mathsf{AccessCount}[i] = 0) = (Lock[i][0] = \perp)$ (I31)

Proof: The only statements that may update $\mathsf{AccessCount}[i]$ or $Lock[i][0]$ are $11[i].p$, $14[i].p$, $28[i].p$, and $39[i].p$, where p is any arbitrary process. After the execution of each of these statements, either both sides of (I31) are true, or both sides of (I31) are false. □

invariant $Winner[i][0] = p \wedge \mathsf{lev}(i) < \mathtt{MAX\_LEVEL} \Rightarrow$

$$
\begin{array}{lr}
(p@\{3, 39..44\} \wedge p.lev < \mathsf{lev}(i)) \vee & \mathcal{A} \\
(p@\{4..8\} \wedge p.break\_level < \mathsf{lev}(i)) \vee & \mathcal{B} \\
(p@\{9..20\} \wedge p.lev < \mathsf{lev}(i)) \vee & \mathcal{C} \\
(p@\{9, 10\} \wedge p.lev = \mathsf{lev}(i)) & \mathcal{D}
\end{array}
$$

(I32)

Proof: The only statement that may establish the antecedent is $40[i].p$. By (I53), $p.lev = \mathsf{lev}(i)$ holds before its execution. In this case, p first executes lines L6, L7, 3b, 3c, and 3e. By (I62), $p.break\_level = 0$ holds throught the execution of these lines. Note that, due to line 3c, $p.lev = \mathsf{lev}(i) - 1$ holds when line 3e is executed. We consider two cases.

First, if $p.lev = 0$ holds when line 3e is executed, then we have $\mathsf{lev}(i) = 1$. In this case, p executes line 3f and establishes $p@\{4\} \wedge p.break\_level = 0 < \mathsf{lev}(i)$, which implies disjunct $\mathcal{B}$.

Second, if $p.lev > 0$ holds when line 3e is executed, then p establishes $p@\{3\} \wedge p.lev = \mathsf{lev}(i) - 1$ by executing line 3e. Thus, disjunct $\mathcal{A}$ is established.

It is straightforward to show the following: **(i)** if disjunct $\mathcal{A}$ is falsified, then disjunct $\mathcal{B}$ is established; **(ii)** if disjunct $\mathcal{B}$ is falsified when $\mathsf{lev}(i) < \mathtt{MAX\_LEVEL}$ holds, then disjunct $\mathcal{C}$ or $\mathcal{D}$ is established; **(iii)** if disjunct $\mathcal{C}$ is falsified when $\mathsf{lev}(i) < \texttt{MAX\_LEVEL}$ holds, then disjunct $\mathcal{D}$ is established.

The only statement that may falsify disjunct $\mathcal{D}$ is $10.p$. By (I58), the antecedent of (I32) implies $\mathsf{Node}(p, \mathsf{lev}(i)) = i$. Hence, if statement $10.p$ is executed when both the antecedent and disjunct $\mathcal{D}$ hold, then by (I53), we have $p.n = i$. This implies that statement $10.p$ falsifies the antecedent. $\square$

invariant $\mathit{Winner}[i][1] = p \;\Rightarrow$

$$\begin{array}{ll}
(p@\{3, 39..44\} \;\wedge\; p.\mathit{lev} < \mathsf{lev}(i)) \;\vee & \mathcal{A}\\
(p@\{4..8\} \;\wedge\; p.\mathit{break\_level} < \mathsf{lev}(i)) \;\vee & \mathcal{B}\\
(p@\{9..20\} \;\wedge\; p.\mathit{lev} < \mathsf{lev}(i)) \;\vee & \mathcal{C}\\
(p@\{9, 15, 16\} \;\wedge\; p.\mathit{lev} = \mathsf{lev}(i)) & \mathcal{D}
\end{array} \tag{I33}$$

Proof: The only statement that may establish the antecedent is $44[i].p$. (The following argument is very similar to that given for $40[i].p$, in the proof of (I32).) By (I53), $p.\mathit{lev} = \mathsf{lev}(i)$ holds before its execution. In this case, p first executes lines L14, L15, 3b, 3c, and 3e. By (I62), $p.\mathit{break\_level} = 0$ holds throught the execution of these lines. Note that, due to line 3c, $p.\mathit{lev} = \mathsf{lev}(i) - 1$ holds when line 3e is executed. We consider two cases.

First, if $p.\mathit{lev} = 0$ holds when line 3e is executed, then we have $\mathsf{lev}(i) = 1$. In this case, p executes line 3f and establishes $p@\{4\} \;\wedge\; p.\mathit{break\_level} = 0 < \mathsf{lev}(i)$, which implies disjunct $\mathcal{B}$.

Second, if $p.\mathit{lev} > 0$ holds when line 3e is executed, then p establishes $p@\{3\} \;\wedge\; p.\mathit{lev} = \mathsf{lev}(i) - 1$ by executing line 3e. Thus, disjunct $\mathcal{A}$ is established.

By (I63), the antecedent implies $\mathsf{lev}(i) < \texttt{MAX\_LEVEL}$. Thus, it is straightforward to show the following: **(i)** if disjunct $\mathcal{A}$ is falsified, then disjunct $\mathcal{B}$ is established; **(ii)** if disjunct $\mathcal{B}$ is falsified when the antecedent holds, then disjunct $\mathcal{C}$ or $\mathcal{D}$ is established; **(iii)** if disjunct $\mathcal{C}$ is falsified when the antecedent holds, then disjunct $\mathcal{D}$ is established.

The only statements that may falsify disjunct $\mathcal{D}$ are $9.p$, $15.p$, and $16.p$. By (I58), the antecedent of (I33) implies $\mathsf{Node}(p, \mathsf{lev}(i)) = i$. Hence, if both the antecedent and disjunct $\mathcal{D}$ hold, then by (I34) and (I53), we have

$$\mathit{Winner}[i][0] \neq p \;\wedge\; p.n = i. \tag{F.1}$$

If statement $9.p$ (respectively, $15.p$) is executed when both the antecedent and disjunct $\mathcal{D}$ hold, then by (F.1), it establishes $p@\{15\}$ (respectively, $p@\{16\}$), preserving disjunct $\mathcal{D}$.

Finally, if statement $16.p$ is executed when both the antecedent and disjunct $\mathcal{D}$ hold, then by (F.1), it falsifies the antecedent. □

invariant $\mathit{Winner}[i][1] = p \ \Rightarrow \ \mathit{Winner}[i][0] \neq p$ (I34)

Proof: The only statement that may establish the antecedent is $44[i].p$. By (I53), $p@\{44\} \ \wedge \ p.n = i$ implies $p.lev = \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$. Thus, by (I32), the consequent holds before and after the execution of $44[i].p$.

The only statement that may falsify the consequent is $40[i].p$. However, by (I33), the antecedent and $p@\{40\}$ imply $p.lev < \mathsf{lev}(i)$, which precludes $p.n = i$ by (I53). □

invariant $\mathit{Waiter}[i] = p \ \Rightarrow$

$$p@\{4..23\} \ \wedge \ p.\mathit{break\_level} = \mathsf{lev}(i) \ \wedge \ p.\mathit{side} = 2 \ \wedge \ (p@\{22, 23\} \ \Rightarrow \ p.n = i) \qquad \text{(I35)}$$

Proof: The only statement that may establish the antecedent is $42[i].p$. In this case, p first executes lines L10, L11, 3b, 3d, and 3e. By (I53), line 3d establishes $p.\mathit{break\_level} = \mathsf{lev}(i)$. Note that $\mathsf{lev}(i) > 0$ holds by definition, since the root node is at level 1. Thus, p exits the **repeat** loop, executes line 3f, and establishes $p@\{4\}$. Moreover, due to line L11, line 3f assigns $p.\mathit{side} := 2$. It follows that statement $42[i].p$ establishes the consequent.

The only statements that may falsify the consequent are $21.p$, $22.p$, and $23.p$. By (I58), the antecedent implies $\mathsf{Node}(p, \mathsf{lev}(i)) = i$. Hence, If statement $21.p$ is executed while $p.\mathit{break\_level} = \mathsf{lev}(i) \ \wedge \ p.\mathit{side} = 2$ holds, then it establishes $p.n = i$, preserving the consequent.

If statement $22.p$ is executed while both the antecedent and the consequent hold, then it establishes $p@\{23\}$, preserving the consequent. Finally, if statement $23.p$ is executed while $p.n = i$ holds, then the antecedent is false after its execution. □

invariant $\mathsf{AccessCount}[i] = 1 \Rightarrow$

$$\begin{array}{lr} (\exists p :: p@\{11, 40\} \ \wedge \ p.n = i \ \wedge \ p.lock[\mathsf{lev}(i)] = Lock[i][0]) \ \vee & \mathcal{A} \\ (\exists p :: \ Winner[i][0] = p \ \wedge \ p.lock[\mathsf{lev}(i)] = Lock[i][0]) \ \vee & \mathcal{B} \\ (\exists q :: q@\{28\} \ \wedge \ q.n = i) & \mathcal{C} \end{array} \tag{I36}$$

Proof: The only statement that may establish the antecedent is $39[i].p$ (where p is any arbitrary process), which may do so only if executed when $\mathsf{AccessCount}[i] = 0$ holds. By (I31), this implies that $Lock[i][0] = \bot$. Also, by (I53), $p.lev = \mathsf{lev}(i)$ holds before and after the execution of $39[i].p$. It follows that disjunct $\mathcal{A}$ is true after the execution of $39[i].p$.

We now consider each disjunct of the consequent in turn.

- **Disjunct $\mathcal{A}$.** Assume that there exists a process p satisfying

$$p@\{11, 40\} \ \wedge \ p.n = i \ \wedge \ p.lock[\mathsf{lev}(i)] = Lock[i][0]. \tag{F.2}$$

The only statement that may falsify (F.2) while the antecedent holds are $11[i].p$ and $40[i].p$ (which may falsify $p@\{11, 40\}$), and $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $Lock[i][0]$), where q is any arbitrary process. (Note that $p.n$ and $p.lock[\mathsf{lev}(i)]$ do not change while $p@\{11, 40\}$ holds.)

The antecedent is false after the execution of statement $11[i].p$. If statement $40[i].p$ is executed when (F.2) holds, then disjunct $\mathcal{B}$ holds after its execution.

The antecedent is false after the execution of either $11[i].q$ or $28[i].q$. If statement $14[i].q$ updates $Lock[i][0]$, then the antecedent is false after its execution. If statement $39[i].q$ is executed when the antecedent holds, then it increments $\mathsf{AccessCount}[i]$, thus falsifying the antecedent.

- **Disjunct $\mathcal{B}$.** Assume that there exists a process p satisfying

$$Winner[i][0] = p \ \wedge \ p.lock[\mathsf{lev}(i)] = Lock[i][0]. \tag{F.3}$$

The only statements that may falsify (F.3) are $2.q$, $10[i].q$, $27[i].q$, $32.q$, and $40[i].q$ (which may update $Winner[i][0]$), $39.p$ (which may update $p.lock[\mathsf{lev}(i)]$), and $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $Lock[i][0]$), where q

is any arbitrary process. Since $\mathsf{AccessCount}[i] = 1$ is established only if i is a non-leaf node, statements $2.q$ and $32.q$ cannot update $\mathit{Winner}[i][0]$.

By (I6), $\mathit{Winner}[i][0] = q$ holds before the execution of $10[i].q$. Taken together with (F.3), we have $p = q$, and hence disjunct $\mathcal{A}$ is true after the execution of statement $10[i].q$.

Disjunct $\mathcal{C}$ is true after the execution of statement $27[i].q$. By (I9), statement $40[i].q$ cannot be executed while (F.3) holds.

Statement $39.p$ may update $p.\mathit{lock}[\mathsf{lev}(i)]$ only if executed when $p.\mathit{lev} = \mathsf{lev}(i)$ holds. Also, by (I53), $p@\{39\}$ implies $p.\mathit{lev} < \texttt{MAX\_LEVEL}$. However, by (I32), these assertions imply $\mathit{Winner}[i][0] \neq p$.

As stated above, each of statements $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ falsifies the antecedent, if executed when the antecedent holds.

- **Disjunct $\boldsymbol{\mathcal{C}}$.** The only statements that may falsify the assertion $q@\{28\} \wedge q.n = i$ is $28[i].q$, which falsifies the antecedent. □

invariant $\mathsf{AccessCount}[i] = 2 \;\Rightarrow$

$$\begin{array}{lr}(\exists p :: p@\{11, 40\} \;\wedge\; p.n = i \;\wedge\; p.\mathit{lock}[\mathsf{lev}(i)] \neq \mathit{Lock}[i][0]) \;\vee & \mathcal{D} \\ (\exists p :: \mathit{Winner}[i][0] = p \;\wedge\; p.\mathit{lock}[\mathsf{lev}(i)] \neq \mathit{Lock}[i][0]) \;\vee & \mathcal{E} \\ (\exists q :: q@\{28\} \;\wedge\; q.n = i) & \mathcal{F}\end{array}$$

(I37)

Proof: The only statement that may establish the antecedent is $39[i].r$ (where r is any arbitrary process), which may do so only if executed when $\mathsf{AccessCount}[i] = 1$ holds. In this case, the consequent of (I36) holds before the execution of $39[i].r$.

First, assume that some process p satisfies disjunct $\mathcal{A}$ of (I36) before the execution of $39[i].r$. Since $p@\{11, 40\}$ holds, we have $r \neq p$. Moreover, $\mathsf{AccessCount}[i] = 1$ implies that there was exactly one *fetch-and-update* invocation on $\mathit{Lock}[i][0]$ since it was last reset to $\bot$. Since the underlying *fetch-and-ϕ* primitive has rank at least three, statement $39[i].r$ must change the value of $\mathit{Lock}[i][0]$. Thus, it establishes disjunct $\mathcal{D}$ of (I37).

Second, assume that some process p satisfies disjunct $\mathcal{B}$ of (I36) before the execution of $39[i].r$. By (I53), $r@\{39\} \;\wedge\; r.n = i$ implies

$$r.\mathit{lev} = \mathsf{lev}(i) < \texttt{MAX\_LEVEL}.$$

Thus, by (I32), we also have

$$p@\{39\} \;\Rightarrow\; p.lev < \mathsf{lev}(i).$$

Combining these two assertions, and using $r@\{39\}$, we again have $r \neq p$. As shown above, if statement $39[i].r$ is executed when $\mathsf{AccessCount}[i] = 1$ holds, then it changes the value of $Lock[i][0]$, and thus establishes disjunct $\mathcal{E}$ of (I37).

If disjunct $\mathcal{C}$ of (I36) holds before the execution of $39[i].r$, then clearly disjunct $\mathcal{F}$ of (I37) holds before and after its execution.

The reasoning for statements that may falsify the consequent is the same as in the proof of (I36). □

The following lemma is used in the proof of several invariants, and also in proving starvation-freedom.

Lemma F.1 Assume that $i = \mathsf{Node}(p, l)$ and $j = \mathsf{Node}(p, l + 1)$ holds for some level l (*i.e.*, node j is a child of node i). Also assume that t and u are consecutive states such that $Winner[j][s] = p \;\wedge\; p@\{3, 4, 39..44\}$ holds at both t and u, and $D_k(p, i, j, s)$ (for some $1 \leq k \leq 7$) holds at state t.

If all of the invariants listed in this appendix hold at t, then $D_{k'}(p, i, j, s)$ holds at state u, for some $k' \geq k$.

Proof: It suffices to consider statements that may falsify each D_k.

The only statements that may falsify D_1 are $25[i].q$ and $29[i].q$, where q is any arbitrary process. Statement $25[i].q$ either preserves D_1 by establishing $q@\{26\} \wedge q.n = i$, or establishes

$$q@\{30\} \;\wedge\; q.n = i \;\wedge\; q.child = ((i-1)\cdot m + 2) \;\wedge\; q.i = 0. \qquad \text{(F.4)}$$

Note that node $(i - 1) \cdot m + 2$ is the leftmost child of node i. Since node j is a child of node i, this in turn implies that $(q.child, q.i) \leq (j, s)$. Thus, statement $25[i].q$ establishes either D_2 or D_3 in this case.

Statement $29[i].q$ also establishes (F.4) by executing lines 29a–29c. Thus, it also establishes either D_2 or D_3.

The only statement that may falsify D_2 is $31[i].q$, where q is a process satisfying $(q.child, q.i) < (j, s)$. Define $(\hat{\jmath}, \hat{s})$ to be the value of $(q.child, q.i)$ before its execution. We consider three cases.

- First, if $\hat{\jmath} = j$, then D_2 implies $\hat{s} = 0 \wedge s = 1$. In this case, statement $31[i].q$ establishes $q@\{30\} \wedge (q.child, q.i) = (\hat{\jmath}, 1) = (j, s)$, and hence D_3 is established.

- Second, if $\hat{\jmath} < j \wedge \hat{s} = 0$ holds, then statement $31[i].q$ establishes $q@\{30\} \wedge (q.child, q.i) = (\hat{\jmath}, 1) < (j, s)$, and hence D_2 is preserved.

- Third, if $\hat{\jmath} < j \wedge \hat{s} = 1$ holds, then since node j is a child of node i, we have $\hat{\jmath} < j \leq (i \cdot m + 1)$. (Note that node $i \cdot m + 1$ is the rightmost child of node i). Thus, statement $31[i].q$ establishes $q@\{30\} \wedge (q.child, q.i) = (\hat{\jmath} + 1, 0) \leq (j, s)$, and hence $D_2 \vee D_3$ holds after its execution.

The only statement that may falsify D_3 is $30[i].q$, where q is a process satisfying $(q.child, q.i) = (j, s)$. Since we assume that $Winner[j][s] = p$ holds, statement $30[i].q$ establishes D_4.

The only statement that may falsify D_4 is $31.q$, where q is a process satisfying $q.proc = p$. In this case, statement $31.q$ establishes D_5.

The only statements that may falsify D_5 are $33.p$ and $35.q$, where q is any arbitrary process. Since we assume that $p@\{3, 4, 39..44\}$ holds, statement $33.p$ cannot be executed. If statement $35.q$ falsifies D_5, then it establishes D_6.

The only statement that may falsify D_6 is $37.q$, where q is a process satisfying $q.proc = p$. In this case, statement $37.q$ establishes D_7.

The only statement that may falsify D_7 is $1.p$, which cannot be executed while $p@\{3, 4, 39..44\}$ holds. $\square$

The following two corollaries are direct consequences of Lemma F.1.

Corollary F.1 Assume that $i = \mathsf{Node}(p, l)$ and $j = \mathsf{Node}(p, l + 1)$ holds for some level l. Also, assume that t and u are consecutive states such that $Winner[j][s] = p \wedge p@\{3, 4, 39..44\}$ holds at both t and u. If $W(p, i, j, s)$ holds at t, then it also holds at u. $\square$

Corollary F.2 Assume that $i = \mathsf{Node}(p, l)$ and $j = \mathsf{Node}(p, l+1)$ holds for some level l. Also, assume that $Winner[j][s] = p \wedge p@\{3, 4, 39..44\} \wedge D_k(p, i, j, s)$ holds at some state t, for some $1 \leq k \leq 7$.

Define h to be 5 if $1 \leq k \leq 5$, and 7 if k is 6 or 7. If $Winner[j][s] = p \wedge p@\{3, 4, 39..44\}$ continues to hold, then $D_h(p, i, j, s)$ is eventually established.

Proof: Assume that $\mathit{Winner}[j][s] = p \ \wedge\ p@\{3, 4, 39..44\}$ continues to hold. Since statements 25–31 contain no busy-waiting loops, each of D_1–D_4 is eventually falsified. (Note that, by (I2), each of D_1–D_4 implies that exactly one process is executing in statements 8–38. Thus, we can consider that process as fixed.) As shown in the proof of Lemma F.1, in this case, D_5 is eventually established.

Similarly, D_6 is eventually falsified, in which case D_7 is established. □

invariant $p@\{41\} \ \wedge\ p.n = i = \mathsf{Node}(p, l) \ \wedge\ j = \mathsf{Node}(p, l+1) \ \wedge$
$\mathit{Winner}[j][s] = p \ \wedge\ \mathit{WaiterLock}[i] = \perp \ \wedge\ i > 1 \ \Rightarrow$
$W(p, i, j, s) \ \vee\ \mathsf{AccessCount}[i] = 2 \ \vee$
$(\exists q : q \neq p :: q@\{12, 41\} \ \wedge\ q.n = i)$ (I38)

Proof: The only statements that may establish the antecedent are $39[i].p$ (which may establish $p@\{41\} \wedge p.n = i$), $2.p$, $40[j].p$, and $44[j].p$ (which may establish $\mathit{Winner}[j][s] = p$), and $24[i].q$ (which may establish $\mathit{WaiterLock}[i] = \perp$), where q is any arbitrary process.

First, consider statement $39[i].p$. Let x be the value of $\mathsf{AccessCount}[i]$ before its execution. We consider four cases.

- If $x = 0$, then by (I31), statement $39[i].p$ establishes $p@\{40\}$. Thus, it cannot establish the antecedent.

- If $x = 1$, then statement $39[i].p$ establishes $\mathsf{AccessCount}[i] = 2$, which implies the consequent.

- Assume $x \geq 2$. Statement $39[i].p$ may establish the antecedent only if

  $$\mathit{Winner}[j][s] = p \ \wedge\ \mathit{WaiterLock}[i] = \perp \ \wedge\ i > 1 \qquad \text{(F.5)}$$

  holds. In this case, by (I39), there exists a process q satisfying $q@\{25..28, 41\} \ \wedge\ q.n = i$. Since $p@\{39\}$, we have $q \neq p$.

  If $q@\{25..28\}$ holds, then D_1 holds before and after the execution of $39[i].p$. Taken together with (F.5), this implies $W(p, i, j, s)$. On the other hand, if $q@\{41\}$ holds, then the last disjunct of the consequent holds before and after the execution of $39[i].p$.

- If $x = \star$, then by (I60), there exists a process q satisfying $q@\{12..14\} \ \wedge \ q.n = i \ \wedge \ q.new \neq \bot$. Since $p@\{39\}$, we have $q \neq p$.

  If $q@\{12\}$ holds, then the last disjunct of the consequent holds before and after the execution of $39[i].p$. On the other hand, if $q@\{13, 14\}$ holds, then by (I40) and (I41), respectively, we have $\mathit{Waiter}[i] \neq \bot$. (Note that the antecedent of (I38) implies $i > 1$.) Hence, by (I14), we have $\mathit{WaiterLock}[i] \neq \bot$, which contradicts the antecedent.

The antecedent is false after the execution of each of $2.p$, $40[j].p$, and $44[j].p$. Statement $24[i].q$ establishes D_1. Taken together with $p@\{41\} \ \wedge \ \mathit{Winner}[j][s] = p$, it follows that $W(p, i, j, s)$ holds after its execution.

By Corollary F.1, $W(p, i, j, s)$ cannot be falsified while the antecedent holds. Hence, the only statements that may falsify the consequent while the antecedent holds are $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $\mathsf{AccessCount}[i]$), and $12[i].q$ and $41[i].q$ (which may falsify $q@\{12, 41\} \ \wedge \ q.n = i$), where q is any arbitrary process, different from p.

Statement $11[i].q$ may potentially falsify (I38) only if executed when both the antecedent and $\mathsf{AccessCount}[i] = 2$ hold. In this case, by applying (I18) with 'p' $\leftarrow q$, we have

$$\mathit{Winner}[i][0] = \bot \quad \wedge \quad \neg(\exists r :: r@\{40\} \ \wedge \ r.n = i).$$

Moreover, by (I2), $q@\{11\}$ implies

$$\neg(\exists r :: r@\{28\}) \quad \wedge \quad (r@\{11\} \ \Rightarrow \ r = q).$$

Combining these assertions, and using (I37), it follows that $q@\{11\} \ \wedge \ q.n = i \ \wedge \ q.lock[\mathsf{lev}(i)] \neq \mathit{Lock}[i][0]$ holds before the execution of $11[i].q$. Since $i > 1$ (by the antecedent) and $q.lev = \mathsf{lev}(i)$ (by (I53)), it follows that statement $11[i].q$ establishes $q@\{12\} \ \wedge \ q.n = i$, thus preserving the consequent.

Statement $14[i].q$ may falsify the consequent only if executed when $q.new \neq \bot$ holds. In this case, by (I19), $\mathsf{AccessCount}[i] = \star$ holds before its execution, and hence statement $14[i].q$ cannot falsify $\mathsf{AccessCount}[i] = 2$.

Statement $28[i].q$ establishes D_1. Hence, if it is executed while the antecedent holds, then $W(p, i, j, s)$ holds after its execution. By (I31), $\mathsf{AccessCount}[i] = 2$ implies $\mathit{Lock}[i][0] \neq \bot$. Thus, if executed when $\mathsf{AccessCount}[i] = 2$ holds, statement $39[i].q$ establishes $q@\{41\} \ \wedge \ q.n = i$, which implies the consequent. (Recall that $q \neq p$.)

Statement $12[i].q$ may falsify the consequent only if executed when $\mathit{Waiter}[i] \neq \bot$. By (I14), this implies $\mathit{WaiterLock}[i] \neq \bot$, and hence the antecedent of (I38) is false before and after its execution. Finally, $\mathit{WaiterLock}[i] \neq \bot$ holds after the execution of statement $41[i].q$, which implies that the antecedent is false. □

invariant $\mathsf{AccessCount}[i] \geq 2 \ \wedge\ \mathit{WaiterLock}[i] = \bot \ \wedge\ i > 1 \ \Rightarrow$
$(\exists p :: p@\{25..28, 41\} \ \wedge\ p.n = i)$ (I39)

Proof: The only statements that may establish the antecedent are $24[i].p$ and $39[i].p$, where p is any arbitrary process. Statement $24[i].p$ establishes $p@\{25\} \ \wedge\ p.n = i$. Statement $39[i].p$ may establish the antecedent only if executed when $\mathsf{AccessCount}[i] = 1$ holds. By (I31), this implies $\mathit{Lock}[i] \neq \bot$. Hence, statement $39[i].p$ establishes $p@\{41\} \ \wedge\ p.n = i$ in this case.

The only statements that may falsify the consequent are $25[i].p$, $28[i].p$, and $41[i].p$. Since $i > 1$, statement $25[i].p$ may falsify the consequent only if $\mathit{Lock}[i][0] = \bot$ holds, which implies that $\mathsf{AccessCount}[i] \geq 2$ is false, by (I31). The antecedent is false after the execution of each of $28[i].p$ and $41[i].p$. □

invariant $p@\{13\} \ \Rightarrow\ \mathit{Waiter}[p.n] \neq \bot$ (I40)

Proof: The only statement that may establish the antecedent is $12.p$, which may do so only if the consequent holds.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statement that may falsify the consequent while the antecedent holds is $23.q$, where q is any arbitrary process. However, by (I2), statement $23.q$ cannot be executed while the antecedent holds. □

invariant $p@\{14\} \ \wedge\ p.n > 1 \ \wedge\ p.\mathit{new} \neq \bot \ \Rightarrow\ \mathit{Waiter}[p.n] \neq \bot$ (I41)

Proof: Let $i = p.n$. The only statements that may establish the antecedent are $11.p$ and $13.p$. Statement $11.p$ may potentially establish the antecedent only if executed when $\mathit{Lock}[i][0] = p.\mathit{lock}[p.\mathit{lev}] \ \wedge\ i > 1$ holds. In this case, by (I53), we have $p.\mathit{lev} = \mathsf{lev}(i)$. Thus, by (I64), we have

$$\mathit{Lock}[i][0] = p.\mathit{lock}[\mathsf{lev}(i)] = \phi(\bot, \alpha_p[p.\mathit{counter}[\mathsf{lev}(i)]]).$$

Since the underlying *fetch-and-ϕ* primitive is self-resettable, statement $11.p$ establishes

$$p.new = \phi\big(\ \phi(\bot, \alpha_p[p.counter[\mathsf{lev}(i)]]),\ \beta_p[p.counter[\mathsf{lev}(i)]]\ \big) = \bot,$$

and hence the antecedent is false after its execution.

By (I40), the consequent is true before and after the execution of statement $13.p$.

Note that $p.n$ cannot change while the antecedent holds. Thus, the only statement that may falsify the consequent while the antecedent holds is $23.q$, where q is any arbitrary process. However, by (I2), statement $23.q$ cannot be executed while the antecedent holds. $\square$

invariant $p@\{42\} \ \wedge\ p.n = i = \mathsf{Node}(p, l) \ \wedge\ j = \mathsf{Node}(p, l+1) \ \wedge$
$\quad Winner[j][s] = p \ \wedge\ i > 1 \ \Rightarrow$
$\quad\quad W(p, i, j, s) \ \vee\ \mathsf{AccessCount}[i] = 2 \ \vee\ Lock[i][1] \neq \bot \ \vee$
$\quad\quad (\exists q : q \neq p :: q@\{12, 18, 19, 41, 43\} \ \wedge\ q.n = i)$ (I42)

Proof: The only statements that may establish the antecedent are $41[i].p$ (which may establish $p@\{42\} \ \wedge\ p.n = i$), and $2.p$, $40[j].p$, and $44[j].p$ (which may establish $Winner[j][s] = p$). Statements $2.p$, $40[j].p$, and $44[j].p$ cannot establish $p@\{42\}$. Statement $41[i].p$ may establish the antecedent only if executed when $Winner[j][s] = p \ \wedge\ WaiterLock[i] = \bot$ holds. In this case, by (I38), the consequent holds before its execution. Clearly, the consequent also holds afterwards. (Note that, by Corollary F.1, statement $41[i].p$ cannot falsify $W(p, i, j, s)$ if executed when $Winner[j][s] = p$ holds.)

By Corollary F.1, $W(p, i, j, s)$ cannot be falsified while the antecedent holds. Hence, the only statements that may falsify the consequent while the antecedent holds are $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $\mathsf{AccessCount}[i]$), $17[i].q$ (which may establish $Lock[i][1] = \bot$), and $12[i].q$, $18[i].q$, $19[i].q$, $41[i].q$, and $43[i].q$ (which may falsify $q@\{12, 18, 19, 41, 43\} \ \wedge\ q.n = i$), where q is any arbitrary process, different from p.

The reasoning for statements $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ is similar to that given in the proof of (I38). Statement $17[i].q$ establishes $q@\{18\} \ \wedge\ q.n = i$.

Statements $12[i].q$ and $19[i].q$ may falsify the consequent only if executed when $Waiter[i] \neq \bot$. Similarly, statements $18[i].q$ and $41[i].q$ may falsify the consequent only if executed when $WaiterLock[i] = \bot$ holds. However, by (I11), neither is possible while the antecedent holds.

Finally, after the execution of $43[i].q$, $Lock[i][1] \neq \bot$ is true, which implies the consequent. $\square$

$$\textbf{invariant}\ p@\{4\} \wedge Waiter[i] = p \wedge \\ i = \mathsf{Node}(p, l) \wedge j = \mathsf{Node}(p, l+1) \wedge \\ Winner[j][s] = p \wedge i > 1 \Rightarrow \\ W(p, i, j, s) \vee \mathsf{AccessCount}[i] = 2 \vee Lock[i][1] \neq \bot \vee \\ (\exists q : q \neq p :: q@\{12, 18, 19, 41, 43\} \wedge q.n = i) \vee \\ (\exists q : q \neq p :: q@\{13, 20\} \wedge q.proc = p) \qquad \text{(I43)}$$

Proof: The only statements that may establish the antecedent are $2.p$ (which may establish $Winner[j][s] = p$) and $40.p$, $42.p$, $43.p$, and $44.p$ (which may establish $p@\{4\}$, $Waiter[i] = p$, or $Winner[j][s] = p$). Statement $2.p$ establishes $p@\{3\}$, and hence cannot establish the antecedent.

By (I35), $Waiter[i] = p$ is false before the execution of each of $40.p$, $42.p$, $43.p$, and $44.p$. Thus, only statement $42[i].p$ may establish the antecedent, which may do so only if $Winner[j][s] = p$ holds. In this case, by (I42), the consequent holds before the execution of $42[i].p$. Clearly, the consequent also holds afterwards. (Note that, by Corollary F.1, statement $42[i].p$ cannot falsify $W(p, i, j, s)$ if executed when $Winner[j][s] = p$ holds.)

By Corollary F.1, $W(p, i, j, s)$ cannot be falsified while the antecedent holds. Hence, the only statements that may falsify the consequent while the antecedent holds are $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ (which may update $\mathsf{AccessCount}[i]$), $17[i].q$ (which may establish $Lock[i][1] = \bot$), $12[i].q$, $18[i].q$, $19[i].q$, $41[i].q$, and $43[i].q$ (which may falsify $q@\{12, 18, 19, 41, 43\} \wedge q.n = i$), and $13.q$ and $20.q$ (which may falsify $q@\{13, 20\} \wedge q.proc = p$), where q is any arbitrary process, different from p.

The reasoning for statements $11[i].q$, $14[i].q$, $28[i].q$, and $39[i].q$ is similar to that given in the proof of (I38). Statement $17[i].q$ establishes $q@\{18\} \wedge q.n = i$.

Statements $12[i].q$ and $19[i].q$ establish $q@\{13, 20\} \wedge q.proc = p$ if executed when the antecedent holds. Statements $18[i].q$ and $41[i].q$ may falsify the consequent only if executed when $WaiterLock[i] = \bot$ holds, which contradicts the antecedent by (I14).

After the execution of $43[i].q$, $Lock[i][1] \neq \bot$ is true, which implies the consequent.

Statements $13[i].q$ and $20[i].q$ may falsify the consequent only if executed when $q.proc = p$ holds, in which case D_5 holds after the execution of either statement. Taken together with the antecedent, this implies $W(p, i, j, s)$. $\square$

$$\textbf{invariant} \quad \big[(p@\{43\} \ \wedge \ p.n = i) \ \vee \\ (p@\{4\} \ \wedge \ p.\mathit{result} = \texttt{SECONDARY\_WAITER} \ \wedge \ p.\mathit{break\_level} = l)\big] \ \wedge \\ i = \mathsf{Node}(p, l) \ \wedge \ j = \mathsf{Node}(p, l+1) \ \wedge \\ \mathit{Winner}[j][s] = p \ \wedge \ \mathit{WaiterLock}[i] = \bot \ \wedge \ i > 1 \ \Rightarrow \\ W(p, i, j, s) \qquad \text{(I44)}$$

Proof: The only statements that may establish the antecedent are $41[i].p$ (which may establish $p@\{43\} \ \wedge \ p.n = i$), $43.p$ (which may establish $p@\{4\} \ \wedge \ p.\mathit{result} = \texttt{SECONDARY\_WAITER} \ \wedge \ p.\mathit{break\_level} = l$), $2.p$, $40[j].p$, and $44[j].p$ (which may establish $\mathit{Winner}[j][s] = p$), and $24[i].q$ (which may establish $\mathit{WaiterLock}[i] = \bot$), where q is any arbitrary process.

After the execution of $41[i].p$, $\mathit{WaiterLock}[i] \neq \bot$ holds, and hence the antecedent is false. Statement $43.p$ may establish the antecedent only if executed when $p.\mathit{lev} = l \ \wedge \ \mathit{Winner}[j][s] = p \ \wedge \ \mathit{WaiterLock}[i] = \bot$. Using $i = \mathsf{Node}(p, l)$, and applying (I53), we also have $p.n = i$, and hence the antecedent of (I44) already holds before the execution of $41[i].p$. It follows that statement $41[i].p$ cannot establish the antecedent.

Statement $2.p$ establishes $p@\{3\}$, and hence cannot establish the antecedent. Statement $40[j].p$ establishes either $p@\{3\}$ or $p@\{4\} \ \wedge \ p.\mathit{result} = \texttt{PRIMARY\_WINNER}$, and hence cannot establish the antecedent.

Statement $24[i].q$ establishes D_1. Taken together with $p@\{4, 43\} \ \wedge \ \mathit{Winner}[j][s] = p$, it follows that $W(p, i, j, s)$ holds after its execution.

By Corollary F.1, the consequent cannot be falsified while $p@\{4, 43\} \ \wedge \ \mathit{Winner}[j][s] = p$ holds. □

$$\textbf{invariant} \quad \big[(p@\{41..43\} \ \wedge \ p.n = 1) \ \vee \ (p@\{4\} \ \wedge \ p.\mathit{break\_level} = 1)\big] \ \wedge \\ j = \mathsf{Node}(p, 2) \ \wedge \ \mathit{Winner}[j][s] = p \ \Rightarrow \\ W(p, 1, j, s) \ \vee \ \mathit{Lock}[1][0] \neq \bot \ \vee \\ (\exists q :: q@\{9..25\} \ \wedge \ q.\mathit{break\_level} = 0) \qquad \text{(I45)}$$

Proof: The only statements that may establish the antecedent are $39[1].p$ (which may establish $p@\{41..43\} \ \wedge \ p.n = 1$), and $2.p$, $40.p$, $42.p$, $43.p$, and $44.p$ (which may establish $p@\{4\} \ \wedge \ p.\mathit{break\_level} = 1$ or $\mathit{Winner}[j][s] = p$). By the semantics of *fetch-and-update*, after the execution of statement $39[1].p$, we have $\mathit{Lock}[1][0] \neq \bot$, which implies the consequent.

Statement $2.p$ establishes $p@\{3\}$, and hence cannot establish the antecedent. By (I62), $p.\mathit{break\_level} = 0$ holds before the execution of either $40.p$ or $44.p$. Since they execute line $\mathsf{3c}$ instead of $\mathsf{3d}$, statements $40.p$ and $44.p$ establish either $p@\{3\}$ or $p@\{4\} \wedge p.\mathit{break\_level} = 0$. Thus, they cannot establish the antecedent.

Statements $42.p$ and $43.p$ may establish the antecedent only if executed when $p.\mathit{lev} = 1 \wedge \mathit{Winner}[j][s] = p$ holds. In this case, by (I53), we also have $p.n = 1$, and hence the antecedent already holds before the execution of either statement. It follows that these statements cannot establish the antecedent.

By applying Corollary F.1 with 'i' $\leftarrow 1$ and 'l' $\leftarrow 1$, it follows that $W(p, 1, j, s)$ cannot be falsified while the antecedent holds. (Note that $\mathsf{Node}(p, 1) = 1$ is true by definition.) Hence, the only statements that may falsify the consequent while the antecedent holds are $11[1].q$, $14[1].q$, and $28[1].q$ (which may establish $\mathit{Lock}[1][0] = \perp$), and $25.q$ (which falsifies $q@\{9..25\}$), where q is any arbitrary process.

By (I53), $q.\mathit{lev} = \mathsf{lev}(1) = 1$ holds before the execution of statements $11[1].q$ and $14[1].q$. Clearly, this is possible only if $q.\mathit{break\_level} = 0$. (See the loop condition for the **for** loop of statements 9–20.) Thus, $q@\{9..25\} \wedge q.\mathit{break\_level} = 0$ holds before and after the execution of either $11[1].q$ or $14[1].q$.

Since statement $28.q$ can be executed only if $q.n > 1$ holds, statement $28.q$ is never executed when $q.n = 1$ holds.

Statement $25.q$ may falsify the consequent only if $q.\mathit{break\_level} = 0$ holds. By (I61), this implies that $q.n = 1$, and hence statement $25.q$ establishes $q@\{30\} \wedge (q.\mathit{child}, q.i) = (2, 0)$. Since node j is a child of the root node, we have $2 \le j \le m + 1$, which in turn implies that $(q.\mathit{child}, q.i) \le (j, s)$. Hence, either $D_2(p, 1, j, s)$ or $D_3(p, 1, j, s)$ holds after the execution of $25.q$. Taken together with the antecedent, it follows that $W(p, 1, j, s)$ holds after its execution. $\square$

invariant $p \in \mathit{WaitingQueue} \Rightarrow p@\{3..33, 39..44\}$ (I46)

Proof: The only statements that may establish the antecedent are $13.q$, $20.q$, $29.q$, and $31.q$ (where q is any arbitrary process), which may do so only if $q.\mathit{proc} = p$ holds. If $q = p$, then clearly the consequent holds before and after the execution of each of these statements. On the other hand, if $q \ne p$, then by applying (I47) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, the consequent holds before and after the execution of each of these statements.

The only statement that may falsify the consequent is $33.p$, which also falsifies the antecedent. $\square$

invariant $p@\{13, 20, 27..29, 31, 36\} \wedge p.\mathit{proc} = q \wedge p \ne q \Rightarrow q@\{3..7, 39..44\}$ (I47)

Proof: The only statements that may establish the antecedent are $12.p$, $19.p$, $26.p$, $30.p$, and $35.p$. Statements $12.p$ and $19.p$ may establish the antecedent only if executed when $\mathit{Waiter}[p.n] = q$ holds, in which case, by applying (I35) with 'p' $\leftarrow q$, we have $q@\{4..23\}$. Taken together with $p@\{12, 19\}$, and using (I2), we have $q@\{4..7\}$, which implies the consequent.

Statements $26.p$ and $30.p$ may establish the antecedent only if executed when $\mathit{Winner}[j][s] = q$, for some node j and $s \in \{0, 1\}$. If $s = 0$, then by (I32) and (I59), we have $q@\{3..32, 39..44\}$. On the other hand, if $s = 1$, then by (I33), we have $q@\{3..20, 39..44\}$. Taken together with $q@\{26, 30\}$, and using (I2), we have the consequent.

Statement $35.p$ may establish the antecedent only if executed when $q \in \mathit{WaitingQueue}$ holds. In this case, by (I46), and by applying (I2) as above, we have the consequent.

The only statement that may falsify the consequent is $7.q$. However, by the correctness of the barrier (statements 7 and 38), statement $7.q$ cannot falsify the consequent while the antecedent holds. □

invariant $\mathit{Promoted} = p \;\Rightarrow\; p@\{3..36, 39..44\}$ (I48)

Proof: The only statement that may establish the antecedent is $36.q$ (where q is any arbitrary process), which may do so only if executed when $q.\mathit{proc} = p$ holds. By (I49), this implies that $q \neq p$. Thus, by applying (I47) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, $q.\mathit{proc} = p$ implies that the consequent holds.

The only statement that may falsify the consequent is $36.p$, which falsifies the antecedent by (I49). □

invariant $p@\{36\} \;\Rightarrow\; p.\mathit{proc} \neq p$ (I49)

Proof: The only statement that may establish the antecedent is $35.p$. By (I46), $p \notin \mathit{WaitingQueue}$ holds before its execution. Thus, statement $35.p$ establishes the consequent. The consequent cannot be falsified while the antecedent holds. □

invariant $\| \mathit{WaitingQueue} \| > 0 \;\Rightarrow$

$$\begin{array}{lr}
(\exists p :: p@\{3, 4, 39..44\} \;\wedge\; \mathit{Spin}[p] = \mathit{true}) \;\vee & \mathcal{A} \\
(\exists p :: p@\{5..35\}) \;\vee & \mathcal{B} \\
(\exists p, q :: p@\{3, 4, 39..44\} \;\wedge\; q@\{36, 37\} \;\wedge\; q.\mathit{proc} = p) & \mathcal{C}
\end{array}$$

(I50)

Proof: The only statements that may establish the antecedent are 13.p, 20.p, 29.p, and 31.p, where p is any arbitrary process. Disjunct $\mathcal{B}$ is true after the execution of each of these statements.

We now consider each disjunct of the consequent in turn.

- **Disjunct $\boldsymbol{\mathcal{A}}$.** Assume that there exists a process p satisfying

$$p@\{3, 4, 39..44\} \;\wedge\; \mathit{Spin}[p] = \mathit{true}. \tag{F.6}$$

  The only statements that may falsify (F.6) are 1.p and 4.p. Statement 1.p cannot be executed while (F.6) holds. Disjunct $\mathcal{B}$ is true after the execution of 4.p.

- **Disjunct $\boldsymbol{\mathcal{B}}$.** The only statement that may falsify disjunct $\mathcal{B}$ is 35.p, where p is any arbitrary process. Statement 35.p establishes

$$p@\{36\} \;\wedge\; p.\mathit{proc} = q, \tag{F.7}$$

  for some q. By (I46), $q@\{3..33, 39..44\}$ holds before the execution of 35.p. Moreover, $p@\{35\}$ implies $p \neq q$, and hence, by (I2), $q@\{3..7, 39..44\}$ holds before and after the execution of 35.p.

  If $q@\{5..7\}$ holds, then q satisfies disjunct $\mathcal{B}$ before and after the execution of 35.p. On the other hand, if $q@\{3, 4, 39..44\}$ holds, then by (F.7), disjunct $\mathcal{C}$ (with 'p' $\leftarrow q$ and 'q' $\leftarrow p$) holds after the execution of 35.p.

- **Disjunct $\boldsymbol{\mathcal{C}}$.** The only statements that may falsify disjunct $\mathcal{C}$ are 4.p and 37.q, where p and q are arbitrary processes satisfying $q.\mathit{proc} = p$. If statement 4.p falsifies disjunct $\mathcal{C}$, then disjunct $\mathcal{B}$ is true after its execution. Statement 37.q establishes $\mathit{Spin}[p] = \mathit{true}$ if executed when $q.\mathit{proc} = p$ holds, and hence also establishes disjunct $\mathcal{A}$. □

invariant $\mathit{Promoted} = p \;\wedge\; \mathit{Spin}[p] = \mathit{false} \;\Rightarrow\; (\exists q :: q@\{37\} \;\wedge\; q.\mathit{proc} = p)$ (I51)

Proof: The only statements that may establish the antecedent are $1.p$ and $36.q$, where q is any arbitrary process. By (I48), $\mathit{Promoted} \neq p$ holds before and after the execution of $1.p$, and hence statement $1.p$ cannot establish the antecedent. If statement $36.q$ establishes the antecedent, then it also establishes the consequent.

The only statement that may falsify the consequent is $37.q$, in which case it also falsifies the antecedent. □

$$\begin{aligned}
\textbf{invariant}\ \ & \big[(p@\{3, 39..44\} \ \wedge\ p.lev < l) \ \vee\ (p@\{4\} \ \wedge\ p.\mathit{break\_level} < l)\big] \ \wedge \\
& i = \mathsf{Node}(p, l) \ \Rightarrow \\
& \qquad \mathit{Winner}[i][0] = p \ \vee\ \mathit{Winner}[i][1] = p \ \vee \\
& \qquad (\exists q :: q@\{28, 29, 36, 37\} \ \wedge\ q.\mathit{proc} = p) \ \vee \\
& \qquad p \in \mathit{WaitingQueue} \ \vee \\
& \qquad \mathit{Spin}[p] = \mathit{true} \qquad\qquad (\text{I52})
\end{aligned}$$

Proof: Throughout the proof of (I52), we assume $i = \mathsf{Node}(p, l)$.

The only statements that may establish the antecedent are $2.p$, $40.p$, $42.p$, $43.p$, and $44.p$. Statement $2.p$ may establish the antecedent only if l equals `MAX_LEVEL`, in which case it also establishes $\mathit{Winner}[i][0] = p$.

Statement $40.p$ decrements $p.lev$ by one (by executing line 3c). Let $\hat{n}$ and $\hat{l}$ be the values of $p.n$ and $p.lev$, respectively, before its execution. If $\hat{l} < l$, then the antecedent already holds before the execution of $40.p$. Thus, assume $\hat{l} \geq l$. Since $l = \mathsf{lev}(i)$ (which follows from $i = \mathsf{Node}(p, l)$), we have $l \geq 1$.

By (I62), $p.\mathit{break\_level} = 0$ holds before the execution of $40.p$. Thus, statement $40.p$ establishes either $p@\{3\} \ \wedge\ p.lev = \hat{l} - 1$ (if $\hat{l} > 1$) or $p@\{4\} \ \wedge\ p.\mathit{break\_level} = 0$ (if $\hat{l} = l = 1$). In either case, statement $40.p$ establishes the antecedent if and only if $\hat{l} = l$, which in turn implies that $\hat{n} = \mathsf{Node}(p, \hat{l}) = i$, by (I53). Thus, if $40.p$ establishes the antecedent, then it also establishes $\mathit{Winner}[i][0] = p$.

A similar argument shows that, if statement $44.p$ establishes the antecedent, then it also establishes $\mathit{Winner}[i][1] = p$.

Statements $42.p$ and $43.p$ may establish the antecedent (by establishing $p@\{4\} \ \wedge\ p.\mathit{break\_level} < l$) only if executed when $p.lev < l$ holds, which implies that the antecedent is already true. Hence, these statements cannot establish the antecedent.

The only statements that may falsify the consequent are $2.q$, $10[i].q$, $27[i].q$, $32.q$, and $40[i].q$ (which may update $\mathit{Winner}[i][0]$), $16[i].q$ and $44[i].q$ (which may update $\mathit{Winner}[i][1]$), $29.q$ and $37.q$ (which may falsify $q@\{28, 29, 36, 37\} \ \wedge\ q.\mathit{proc} = p$), $33.p$

and 35.q (which may falsify $p \in \mathit{WaitingQueue}$), and 1.$p$ (which falsifies $\mathit{Spin}[p] = \mathit{true}$), where q is any arbitrary process.

Statements 2.q and 32.q update q's dedicated leaf node. Since $i = \mathsf{Node}(p, l)$ (*i.e.*, i is a node on p's path), clearly i is not q's leaf node. Thus, these statements cannot update $\mathit{Winner}[i][0]$.

By (I6), $\mathit{Winner}[i][0] = q$ holds before the execution of 10[i].q. Therefore, statement 10[i].q may falsify the consequent only if $p = q$, in which case the antecedent is false before and after its execution.

By (I22), if statement 27[i].q falsifies $\mathit{Winner}[i][0] = p$, then it establishes $q@\{28\} \wedge q.\mathit{proc} = p$.

By (I9), $\mathit{Winner}[i][0] = p$ is false before the execution of 40[i].q. Thus, statement 40[i].q cannot falsify $\mathit{Winner}[i][0] = p$.

By (I7), $\mathit{Winner}[i][1] = q$ holds before the execution of 16[i].q. Therefore, statement 16[i].q may falsify the consequent only if $p = q$, in which case the antecedent is false before and after its execution.

By (I10), $\mathit{Winner}[i][1] = p$ is false before the execution of 44[i].q. Thus, statement 44[i].q cannot falsify $\mathit{Winner}[i][1] = p$.

Statements 29.q and 37.q may falsify the consequent only if executed when $q.\mathit{proc} = p$, in which case they establish either $p \in \mathit{WaitingQueue}$ or $\mathit{Spin}[p] = \mathit{true}$. Thus, these statements preserve the consequent.

Statements 1.p and 33.p cannot be executed while the antecedent holds. Finally, if statement 35.q falsifies the consequent, then it establishes $q@\{36\} \wedge q.\mathit{proc} = p$, and hence preserves the consequent. □

invariant $p@\{9..20, 39..44\} \Rightarrow$

$$p.n = \mathsf{Node}(p, p.\mathit{lev}) \wedge p.\mathit{lev} = \mathsf{lev}(p.n) < \texttt{MAX\_LEVEL} \qquad \text{(I53)}$$

Proof: Whenever the antecedent is established, p executes either line 8c or L1. Hence, $p.n = \mathsf{Node}(p, p.\mathit{lev})$ is also established, which also implies $p.\mathit{lev} = \mathsf{lev}(p.n)$. Moreover, the loop condition of the **for** loop of statements 9–20 ensures that, if $p@\{9..20\}$ holds, then $p.\mathit{lev} < \texttt{MAX\_LEVEL}$ also holds. Similarly, lines 2b and 3c ensure that, if $p@\{39..44\}$ holds, then $p.\mathit{lev} < \texttt{MAX\_LEVEL}$ also holds.

The consequent cannot be falsified while the antecedent holds. □

invariant $p@\{4..38\} \Rightarrow \big[(p.\mathit{break\_level} > 0) = (p.\mathit{side} = 2)\big]$ (I54)

Proof: The only statements that may establish the antecedent are $40.p$, $42.p$, $43.p$, and $44.p$. If p executes either $40.p$ or $44.p$, then p establishes the antecedent if and only if it executes lines 3b, 3c, 3e, and 3f, in which case line 3f establishes $p.side \neq 2$. Moreover, by (I62), $p.break\_level = 0$ holds before and after the execution of either $40.p$ or $44.p$. Thus, in this case, the consequent holds after the execution of either $40.p$ or $44.p$.

By (I62), $p.lev > 0$ holds before the execution of either $42.p$ or $43.p$. Thus, statement $42.p$ establishes $p.break\_level > 0 \wedge p.side = 2$ by executing lines L10, L11, 3b, and 3d–3f. Therefore, the consequent holds after its execution. Similarly, statement $43.p$ may establish the antecedent only if it executes lines L12, L13, L16, 3b, and 3d–3f, in which case it also establishes $p.break\_level > 0 \wedge p.side = 2$, and hence, the consequent.

The consequent cannot be falsified while the antecedent holds. □

invariant $p@\{35, 36\} \Rightarrow Promoted = p \vee Promoted = \perp$ (I55)
invariant $p@\{37\} \Rightarrow p.proc = Promoted$ (I56)

Proof: Note that the only statement that may update *Promoted* is statement 36. Since executions of statements 8–38 are serialized by the barrier (see (I2)), invariants (I55) and (I56) follow easily by inspecting statements 34–36. □

invariant $p@\{22..31\} \Rightarrow \mathsf{lev}(p.n) < \mathtt{MAX\_LEVEL}$ (I57)

Proof: The only statement that may establish the antecedent is $21.p$.

Note that process p may update $p.break\_level$ only via the execution of line 2b or 3d. Clearly, $p.break\_level < \mathtt{MAX\_LEVEL}$ holds after the execution of line 2b. Since line 3d is always executed as a part of statement $42.p$ or $43.p$, by (I53), $p.break\_level < \mathtt{MAX\_LEVEL}$ also holds after the execution of line 3d.

It follows that $p.break\_level < \mathtt{MAX\_LEVEL}$ holds before the execution of statement $21.p$. Hence, the consequent holds after its execution.

The consequent cannot be falsified while the antecedent holds. □

invariant $Winner[i][s] = p \vee Waiter[i] = p \Rightarrow i = \mathsf{Node}(p, \mathsf{lev}(i))$ (I58)

Proof: Since process p visits only the nodes on its path (*i.e.*, $\mathsf{Node}(p, l)$ for $1 \leq l \leq \mathtt{MAX\_LEVEL}$), this invariant follows easily. □

invariant $\mathit{Winner}[i][0] = p \ \wedge\ \mathsf{lev}(i) = \texttt{MAX\_LEVEL} \ \Rightarrow\ p@\{3..32, 39..44\}$ (I59)

Proof: The only statements that may establish the antecedent are $2.p$ and $40[i].p$. Statement $2.p$ establishes the consequent. By (I53), statement $40[i].p$ cannot be executed if $\mathsf{lev}(i) = \texttt{MAX\_LEVEL}$.

The only statement that may falsify the consequent is $32.p$. By (I58), the antecedent implies $i = \mathsf{Node}(p, \texttt{MAX\_LEVEL})$. Thus, statement $32.p$ falsifies the antecedent. □

invariant $\mathsf{AccessCount}[i] = \star \ \Rightarrow\ (\exists p :: p@\{12..14\} \ \wedge\ p.n = i \ \wedge\ p.\mathit{new} \neq \perp)$ (I60)

Proof: The only statement that may establish the antecedent is $11[i].p$, where p is any arbitrary process. If statement $11[i].p$ establishes the antecedent, then it also establishes the consequent.

The only statement that may falsify the consequent is $14[i].p$ (where p is any arbitrary process), which may do so only if executed when $p.\mathit{new} \neq \perp$ holds. In this case, statement $14[i].p$ also falsifies the antecedent. □

invariant $p@\{22..31\} \ \wedge\ p.\mathit{break\_level} = 0 \ \Rightarrow\ p.n = 1$ (I61)

Proof: The only statement that may establish the antecedent is $21.p$, which may do so only if $p.\mathit{break\_level} = 0$ holds. By (I54), this implies $p.\mathit{side} \neq 2$. Thus, statement $21.p$ establishes the consequent in this case.

The consequent cannot be falsified while the antecedent holds. □

invariant $p@\{3, 39..44\} \ \Rightarrow\ p.\mathit{lev} > 0 \ \wedge\ p.\mathit{break\_level} = 0$ (I62)

Proof: This invariant follows easily by inspecting lines 2b, 3c, 3d, and 3e. □

invariant $\mathit{Winner}[i][1] \neq \perp \ \Rightarrow\ \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$ (I63)

Proof: The only statement that may establish the antecedent is $44[i].p$. By (I53), statement $44[i].p$ may be executed only if the consequent is true. □

invariant $(\mathit{Winner}[i][0] = p \ \wedge\ \mathsf{lev}(i) < \texttt{MAX\_LEVEL}) \ \vee$
$(p@\{11, 40\} \ \wedge\ p.n = i) \ \Rightarrow$
$p.\mathit{lock}[\mathsf{lev}(i)] = \phi(\perp, \alpha_p[p.\mathit{counter}[\mathsf{lev}(i)]])$ (I64)

Proof: The only statements that may establish the antecedent are $40[i].p$ (which may establish $Winner[i][0] = p \ \wedge\ \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$), and $10[i].p$ and $39[i].p$ (which may establish $p@\{11, 40\} \ \wedge\ p.n = i$). However, the antecedent is already true (specifically, its second disjunct) before the execution of $40[i].p$. Similarly, by (I6) and (I53), the antecedent is already true before the execution of $10[i].p$. Hence, these statements cannot establish the antecedent.

Statement $39[i].p$ may establish the antecedent only if executed when $Lock[i][0] = \bot$ holds. By (I53), $p@\{39\} \ \wedge\ p.n = i$ implies $p.lev = \mathsf{lev}(i)$, and hence statement $39[i].p$ establishes the consequent in this case.

The only statement that may falsify the consequent while the antecedent holds is $39.p$ (which may update $p.lock[\mathsf{lev}(i)]$ and $p.counter[\mathsf{lev}(i)]$), which may do so only if $p.lev = \mathsf{lev}(i)$ holds. Taken together with the antecedent, we have

$$Winner[i][0] = p \ \wedge\ p.lev = \mathsf{lev}(i) < \texttt{MAX\_LEVEL} \ \wedge\ p@\{39\}.$$

However, this is precluded by (I32). $\square$

invariant $Lock[i][1] \neq \bot \ \Rightarrow\ (\exists p :: p@\{17, 44\} \ \wedge\ p.n = i) \ \vee\ Winner[i][1] \neq \bot$ (I65)

Proof: The only statement that may establish the antecedent is $43[i].p$, where p is any arbitrary process. However, if statement $43[i].p$ is executed when the antecedent is false, then it establishes $p@\{44\} \ \wedge\ p.n = i$, which implies the consequent.

The only statements that may falsify the consequent are $16[i].p$ (which may falsify $Winner[i][1] \neq \bot$) and $17[i].p$ and $44[i].p$ (which may falsify the first disjunct of the consequent), where p is any arbitrary process. Statement $16[i].p$ establishes $p@\{17\} \ \wedge\ p.n = i$. Statement $17[i].p$ falsifies the antecedent. Statement $44[i].p$ establishes $Winner[i][1] \neq \bot$. $\square$

invariant $p@\{4\} \ \wedge\ p.result = \texttt{PRIMARY\_WAITER} \ \Rightarrow$
$Waiter[\mathsf{Node}(p, p.break\_level)] = p$ (I66)

Proof: The only statement that may establish the antecedent is $42.p$. By (I53), $p.n = \mathsf{Node}(p, p.lev)$ holds before its execution. Thus, statement $42.p$ establishes the consequent (via the execution of lines L10 and 3d).

Let $i = \mathsf{Node}(p, p.break\_level)$. Note that $p.break\_level$ cannot be updated while the antecedent holds. Hence, the only statements that may falsify the consequent while the antecedent holds are $23[i].q$ and $42[i].q$, where q is any arbitrary process. By (I8), statement $23[i].q$ may falsify the consequent only if $q = p$. In this case, the antecedent is false before and after its execution.

By (I11), statement $42[i].q$ cannot be executed while the consequent holds. □

This completes the proof of the Exclusion property.

F.3 Proof of Starvation-freedom

In order to prove starvation-freedom, we must show that each busy-waiting loop at statements 4, 12, 19, and 26 eventually terminates. Toward this goal, we first prove the following lemma. (Statement 4 is considered later.)

Lemma F.2 The busy-waiting loops at statements 12, 19, and 26 each eventually terminate.

Proof: Statement 12: If a process p is busy-waiting at statement $12[i]$, then by (I26), either there exists a process q satisfying $q@\{41\} \wedge q.n = i$, or $\mathit{WaiterLock}[i] \neq \perp$ holds. In the former case, q eventually executes statement $41[i]$. Thus, in either case, $\mathit{WaiterLock}[i] \neq \perp$ is eventually established at some state t.

At state t, by (I28), either there exists a process r satisfying $r@\{24, 42\} \wedge r.n = i$, or $\mathit{Waiter}[i] \neq \perp$ holds. Since $p@\{12\}$ holds, $r@\{24\}$ is precluded by (I2). Moreover, if $r@\{42\} \wedge r.n = i$ holds, then r eventually executes statement $42[i]$. Thus, in either case, $\mathit{Waiter}[i] \neq \perp$ is eventually established at some later state u.

Finally, $\mathit{Waiter}[i] \neq \perp$ may be falsified only by statement $23.q$, where q is any arbitrary process. By (I2), statement $23.q$ cannot be executed while $p@\{12\}$ holds. It follows that $\mathit{Waiter}[i] \neq \perp$ holds continuously from state u, and hence p eventually establishes $p@\{13\}$ by executing $12.p$.

Statement 19: Assume that, at state t, a process p is busy-waiting at statement $19[i]$. By (I27), $\mathit{WaiterLock}[i] \neq \perp$ holds at state t. The rest of the reasoning is the same as for statement 12.

Statement 26: If a process p is busy-waiting at statement $26[i]$, then by (I29) and (I57), we have $\mathit{Lock}[i][0] \neq \perp \wedge \mathsf{lev}(i) < \texttt{MAX\_LEVEL}$. Thus, by (I2) and (I30), and using $p@\{26\}$, we have the following: either there exists a process q satisfying $q@\{40\} \wedge q.n = i$, or $\mathit{Winner}[i][0] \neq \perp$ holds. In the former case, q eventually executes statement $40[i]$. Thus, in either case, $\mathit{Winner}[i][0] \neq \perp$ is eventually established.

$Winner[i][0] \neq \bot$ may be falsified only by statements $10.q$, $27.q$, and $32.q$, where q is any arbitrary process. By (I2), none of these statements can be executed while $p@\{26\}$ holds. It follows that $Winner[i][0] \neq \bot$ holds continuously, and hence p eventually establishes $p@\{27\}$ by executing $26.p$. □

We now prove that the busy-waiting loop at statement 4 eventually terminates. Toward this goal, we first prove the following *leads-to* properties. (See page 385 for the definition of A *leads-to* B.) Property (L1) states that, if a process p, enqueued onto *WaitingQueue*, is waiting at statement 4, then eventually some process q executes statement 35 and dequeues a process from *WaitingQueue*.

$$p@\{4\} \;\wedge\; p \in \mathit{WaitingQueue} \quad \textit{leads-to} \quad p@\{5\} \;\vee\; (\exists q : q \neq p :: q@\{35\}) \qquad \text{(L1)}$$

Proof: If $p@\{5\}$ is eventually established, then (L1) is proven. Thus, in the rest of the proof, we assume that $p@\{4\}$ holds indefinitely.

Since the left-hand side of (L1) implies $\|\mathit{WaitingQueue}\| > 0$, by (I50), one of the following holds, for some processes q and r.

- $\mathcal{A}$: $q@\{3, 4, 39..44\} \;\wedge\; r@\{36, 37\} \;\wedge\; r.proc = q$,
- $\mathcal{B}$: $q@\{3, 4, 39..44\} \;\wedge\; Spin[q] = true$,
- $\mathcal{C}$: $q@\{5..33\}$, or
- $\mathcal{D}$: $q@\{34, 35\}$.

First, if $\mathcal{A} \wedge \neg\mathcal{B}$ is true, then eventually r establishes $\mathcal{B}$ by executing statement $37.r$.

Note that $Spin[q] = true$ can be falsified only by statement $1.q$. Hence, $Spin[q] = true$ cannot be falsified while $\mathcal{B}$ holds. Therefore, if $\mathcal{B}$ is true, then eventually q establishes $\mathcal{C}$ by executing statement $4.q$.

If $\mathcal{C}$ is true, then by Lemma F.2, q eventually establishes $\mathcal{D}$ by executing statement $33.q$.

It follows that, in all cases, $\mathcal{D}$ is eventually established. Since $p@\{4\}$ continues to hold, we have $q \neq p$. Hence, if $q@\{35\}$ holds, we have the right-hand side of (L1). Thus, assume that $q@\{34\}$ holds. Eventually, q executes statement $34.q$. Let t be the state just before the execution of statement $34.q$. We consider two cases.

First, if $Promoted \in \{q, \bot\}$ holds at state t, then q establishes $q@\{35\}$, which implies the right-hand side of (L1).

Otherwise, $Promoted = r \neq \bot$ holds for some process $r \neq q$. By (I2), $q@\{35\}$ implies $(\forall r' :: \neg r'@\{37\})$. Thus, by applying (I51) with 'p' $\leftarrow r$, we have $Spin[r] = true$.

Also, by applying (I48) with 'p' $\leftarrow r$, we have $r@\{3..36, 39..44\}$. By (I2) and $q@\{35\}$, this in turn implies that

- $r@\{3..7, 39..44\}$ holds at state t. (F.8)

Note that $Spin[r] = true$ cannot be falsified while $r@\{3..7, 39..44\}$ holds. Thus, r makes progress at statement 4, and eventually establishes $r@\{7\}$. After that, by Lemma F.2, r executes statements 7–33 and eventually establishes $r@\{34\}$.

We claim that $Promoted = r$ is not falsified until r executes statement 34. The only statement that may falsify $Promoted = r$ is $36.r'$, where r' is any arbitrary process. However, by (I55), $36.r'$ may falsify $Promoted = r$ only if $r' = r$. Also, by (F.8), statement $36.r$ cannot be executed (after state t) until r executes statement $34.r$.

It follows that $Promoted = r$ holds when r executes statement 34. Hence, r establishes $r@\{35\}$. By our assumption that $p@\{4\}$ holds indefinitely, we have $r \neq p$, and hence $r@\{35\}$ implies the right-hand side of (L1). □

The following property is a simple application of (L1).

$$p@\{4\} \ \wedge \ p \in \mathit{WaitingQueue} \quad \mathit{leads\text{-}to} \quad p@\{5\} \ \vee \ \mathit{Spin}[p] = \mathit{true} \tag{L2}$$

Proof: If $p@\{5\}$ is eventually established, then (L2) holds. Thus, assume that $p@\{4\}$ holds indefinitely. In this case, by (L1), some process q_1 eventually executes statement 35, and dequeues a process r from *WaitingQueue*. If $r = p$, then q_1 establishes $Spin[p] = true$ by executing statement 37. Otherwise, the left-hand side of (L2) continues to hold, and hence some process q_2 eventually executes statement 35. Since the queue is finite, continuing in this manner, eventually some process q_i (for some i) dequeues process p by executing statement 35, and then establishes $Spin[p] = true$ at statement 37. □

We now prove the following two *leads-to* properties. In proving (L3) and (L4), we use induction on l. In particular, when we prove (L3) and (L4) for a particular value of l, we assume that (L3) and (L4) hold for all smaller values of l. In addition, when we prove (L4), we assume that (L3) holds for the same value of l.

$$\begin{aligned} &p@\{4\} \ \wedge \ p.\mathit{side} = 2 \ \wedge \ p.\mathit{break\_level} = l \quad \mathit{leads\text{-}to} \\ &\qquad p \in \mathit{WaitingQueue} \ \vee \ \mathit{Spin}[p] = \mathit{true} \ \vee \ p@\{5\} \end{aligned} \tag{L3}$$

$$p@\{4\} \ \wedge \ p.\mathit{side} = 2 \ \wedge \ p.\mathit{break\_level} = l \quad \mathit{leads\text{-}to} \quad p@\{5\} \tag{L4}$$

Proof of (L3): By (I54), it suffices to assume $p.break\_level > 0$. Consider a state t in which the left-hand side of (L3) is true.

Let

$$i = \mathsf{Node}(p, l) \quad \wedge \quad j = \mathsf{Node}(p, l+1). \tag{F.9}$$

By applying (I52) with 'l' $\leftarrow l+1$ and 'i' $\leftarrow j$, it follows that at state t and at every subsequent state until $p@\{4\}$ is falsified, one of the following propositions holds.

(P1) $Winner[j][0] = p \ \vee \ Winner[j][1] = p$,

(P2) there exists a process q satisfying $q@\{28, 29, 36, 37\} \ \wedge \ q.proc = p$,

(P3) $p \in WaitingQueue$, and

(P4) $Spin[p] = true$.

If either (P3) or (P4) is true at some subsequent state, then (L3) holds. If (P2) is true at some subsequent state, then eventually q executes either statement 29 or 37, establishing (P3) or (P4).

The only remaining case is when (P1) $\wedge$ $\neg$(P2) is true at state t and at *all subsequent states*. In this case,

- $Winner[j][s] = p$ holds at state t, for some $s \in \{0, 1\}$. (F.10)

For the sake of contradiction, we further assume that the right-hand side of (L3) is *not* eventually established. Hence, at t and at all subsequent states, we have the following.

$$\neg(\exists q :: q@\{28, 29, 36, 37\} \ \wedge \ q.proc = p), \tag{F.11}$$

$$p \notin WaitingQueue, \tag{F.12}$$

$$Spin[p] = false, \quad \text{and} \tag{F.13}$$

$$p@\{4\}. \tag{F.14}$$

Our goal now is to derive a contradiction from (F.11)–(F.14). Throughout the rest of the proof, all propositions are assumed to apply to state t and all subsequent states, unless stated otherwise. We start with the following claims.

Claim 1: $Winner[j][s] = p$.

Proof of Claim: At state t, $Winner[j][s] = p$ holds by (F.10). The only statements that may falsify $Winner[j][s] = p$ are $2.q$, $10[j].q$, $27[j].q$, $32.q$, and $40[j].q$ (if $s = 0$), and $16[j].q$ and $44[j].q$ (if $s = 1$), where q is any arbitrary process. By (F.14), we have $q \neq p$.

Statements $2.q$ and $32.q$ update q's dedicated leaf node. Since $j = \mathsf{Node}(p, l+1)$ by (F.9) (*i.e.*, j is a node in p's path), clearly j is not q's leaf node. Thus, these statements cannot update $\mathit{Winner}[j][0]$.

By (I6), $\mathit{Winner}[j][0] = q$ holds before the execution of $10[j].q$. Thus, because $q \neq p$, $\mathit{Winner}[j][0] = p$ is already false prior to its execution.

By (I22), if statement $27[j].q$ falsifies $\mathit{Winner}[j][0] = p$, then it establishes $q@\{28\} \ \wedge \ q.\mathit{proc} = p$, which contradicts (F.11).

By (I9), $\mathit{Winner}[j][0] = p$ is false before the execution of $40[j].q$. Thus, statement $40[j].q$ cannot falsify $\mathit{Winner}[j][0] = p$.

By (I7), $\mathit{Winner}[j][1] = q$ holds before the execution of $16[j].q$. Thus, because $q \neq p$, $\mathit{Winner}[j][1] = p$ is already false prior to its execution.

By (I10), $\mathit{Winner}[j][1] = p$ is false before the execution of $44[j].q$. Thus, statement $44[j].q$ cannot falsify $\mathit{Winner}[j][1] = p$. □

Claim 2: For all k ($1 \leq k \leq 7$), $\neg D_k(p, i, j, s)$. (In particular, $\neg W(p, i, j, s)$.)

Proof of Claim: If $D_k(p, i, j, s)$ at some state u, then by (F.14) and Claim 1, $W(p, i, j, s)$ holds. Hence, by Corollary F.2, $p \in \mathit{WaitingQueue} \ \vee \ \mathit{Spin}[p] = \mathit{true}$ is eventually established. But this contradicts (F.12) and (F.13). □

We now consider three cases, depending on the value of l.

Case 1: $l = 1$. Since $l = 1$, we have $i = 1$ and $j = \mathsf{Node}(p, 2)$. By (I45) and Claims 1 and 2, at state t and at all subsequent states, we have one of the following.

(P5) $\mathit{Lock}[1][0] \neq \bot$, and
(P6) there exists a process q satisfying $q@\{9..25\} \ \wedge \ q.\mathit{break\_level} = 0$.

We first prove the following claim.

Claim 3: (P5) *leads-to* (P6).

Proof of Claim: Assume that (P5) holds at some state. By applying (I30) with 'i' $\leftarrow 1$, either there exists a process q satisfying

$$q@\{11..14, 28, 40\} \ \wedge \ q.n = 1, \tag{F.15}$$

or $Winner[1][0] \neq \perp$ holds. In the latter case, by applying (I32) with 'i' $\leftarrow$ 1, and using $\mathsf{lev}(1) = 1$, it follows that there exists a process q satisfying

$$q@\{4..8\} \ \wedge \ q.break\_level = 0, \quad \text{or} \tag{F.16}$$

$$q@\{9, 10\} \ \wedge \ q.lev = 1. \tag{F.17}$$

(Note that $q.lev < 1$ is always false while $q@\{9..20\}$ holds; hence, the other two disjuncts in the consequent of (I32) are precluded.)

Since statement $28.q$ can be executed only if $q.n > 1$ holds, (F.15) implies $q@\{11..14, 40\} \ \wedge \ q.n = 1$. Also, by (I53), (F.17) implies $q.n = 1$. Combining these assertions, there exists a process q satisfying one of the following.

(P7) $q@\{40\} \ \wedge \ q.n = 1$,
(P8) $q@\{4..8\} \ \wedge \ q.break\_level = 0$, and
(P9) $q@\{9..14\} \ \wedge \ q.n = 1$.

We claim that, if any of (P7)–(P9) holds, then eventually (P6) holds. First, assume that (P7) holds at some state. By (I53) and (I62), (P7) implies $q.lev = 1 \ \wedge \ q.break\_level = 0$. Hence, eventually q establishes (P8) by executing statement 40.

Second, assume that (P8) holds at some state. By (I54), (P8) implies $p.side \neq 2$, and hence q does not busy-wait at statement 4. It follows that q eventually establishes (P9).

Finally, if (P9) holds at some state, then by (I53), we also have $q.lev =$ 1. Due to the loop condition of **for** loop at lines 9–20, this implies that $q.break\_level = 0$. This in turn implies (P6). □

By Claim 3, in all cases, eventually (P6) is established. Hence, by Lemma F.2, eventually the following holds: $q@\{25\} \ \wedge \ q.break\_level = 0$. By (I61), this in turn implies $q.n = 1$. However, $q@\{25\} \wedge q.n = 1$ implies $D_1(p, i, j, s)$, which contradicts Claim 2. Thus, we have reached contradiction.

Case 2: $l > 1 \wedge p.result =$ `PRIMARY_WAITER`. In this case, since $l > 1$, we have

$$i > 1. \tag{F.18}$$

By (I66), we also have

$$\mathit{Waiter}[i] = p. \tag{F.19}$$

Hence, by (I43) and Claims 1 and 2, at state t and all subsequent states, one of the following holds.

(P10) $\mathsf{AccessCount}[i] = 2$,
(P11) $(\exists q : q \neq p :: q@\{41, 43\} \wedge q.n = i)$,
(P12) $\mathit{Lock}[i][1] \neq \perp$,
(P13) $(\exists q : q \neq p :: q@\{12, 18, 19\} \wedge q.n = i)$, and
(P14) $(\exists q : q \neq p :: q@\{13, 20\} \wedge q.proc = p$.

We claim that, in all cases, eventually (P14) holds. Toward this goal, we prove the following claims. (Recall that A *unless* B is true if and only if the following holds: if $A \wedge \neg B$ holds before some statement execution, then $A \vee B$ holds after that execution. Informally, A is not falsified until B is established.)

Claim 4: (P10) *unless* (P11) $\vee$ (P13).

Proof of Claim: It suffices to consider statements that may falsify (P10). Consider a state u at which (P10) holds.

By Claim 2, for each process q, we have $\neg(q@\{28\} \wedge q.n = i)$. (Otherwise, $D_1(p, i, j, s)$ would hold.) Therefore, by (I37), there exists a process q satisfying the following:

$$q.lock[\mathsf{lev}(i)] \neq \mathit{Lock}[i][0], \quad \text{and} \tag{F.20}$$

$$(q@\{11, 40\} \wedge q.n = i) \vee \mathit{Winner}[i][0] = q. \tag{F.21}$$

The only statements that may falsify (P10) are $11[i].r$, $14[i].r$, $28[i].r$, and $39[i].r$, where r is any arbitrary process.

Before the execution of either statement $11[i].r$ or $28[i].r$, by (I18) (with 'p' $\leftarrow r$) and (F.21), $q@\{11\}$ must hold. Thus, by (I2), statement $28[i].r$ cannot be executed while (P10) holds, and statement $11[i].r$ may be executed only if $r = q$. In the latter case, by (F.18) and (F.20), statement $11[i].r$ establishes $q@\{12\} \wedge q.n = i$, which in turn implies (P13). (Note that $q@\{12\}$ implies $q \neq p$ by (F.14).)

Statement $14[i].r$ may falsify (P10) only if executed when $r.new \neq \perp$ holds. In this case, by (I19) (with 'p' $\leftarrow r$) and (F.21), $q@\{11\}$ must hold. However, this contradicts $r@\{14\}$ by (I2).

Finally, Before the execution of statement $39[i].r$, by (I31), (P10) implies $Lock[i][0] \neq \perp$. Thus, statement $39[i].r$ establishes $r@\{41\} \wedge r.n = i$, which in turn implies (P11) (with 'q' $\leftarrow r$). (Note that $r@\{41\}$ implies $r \neq p$ by (F.14).) □

Claim 5: (P10) *leads-to* (P11) $\vee$ (P13).

Proof of Claim: Due to Claim 4, it suffices to show that (P10) is eventually falsified. As shown in the proof of Claim 4, (P10) implies that there exists a process q satisfying (F.21). Thus, we have one of the following.

$\mathcal{A}$: $q@\{40\} \wedge q.n = i$,
$\mathcal{B}$: $Winner[i][0] = q$, and
$\mathcal{C}$: $q@\{11\} \wedge q.n = i$.

We claim that q eventually executes $11[i].q$, thus falsifying (P10). First, if $\mathcal{A}$ holds, then eventually q executes statement $40[i]$, establishing $\mathcal{B}$.

Second, assume that $\mathcal{B}$ holds. We establish the following claims.

Claim 5-1: $\mathcal{B}$ *unless* $\mathcal{C}$.

Proof of Claim: The only statements that may falsify $\mathcal{B}$ are $2.r$, $10[i].r$, $27[i].r$, $32.r$, and $40[i].r$, where r is any arbitrary process. Statements $2.r$ and $32.r$ update r's dedicated leaf node. Since $i = \mathsf{Node}(p, l)$ by (F.9) (*i.e.*, i is a node in p's path), clearly i is not r's leaf node. Thus, these statements cannot update $Winner[i][0]$. By (I6), $Winner[i][0] = r$ holds before the execution of $10[i].r$. Thus, statement $10[i].r$ can falsify $\mathcal{B}$ only if $r = q$, in which case it establishes $\mathcal{C}$.

Before the execution of $27[i].r$, $D_1(p, i, j, s)$ holds by definition, which contradicts Claim 2. Hence, statement $27[i].r$ cannot be executed.

By (I9), $Winner[i][0] = \bot$ holds before the execution of $40[i].r$. Thus, statement $40[i].r$ cannot falsify $\mathcal{B}$. $\Box$

Claim 5-2: $\mathcal{B}$ *leads-to* $\mathcal{C}$.

Proof of Claim: Due to Claim 5-1, it suffices to show that $\mathcal{B}$ is eventually falsified. By (I32), $\mathcal{B}$ implies that one of the following holds for some process q. (Note that, since node i has a child node j, we have $\mathsf{lev}(i) < \texttt{MAX\_LEVEL}$.)

$\mathcal{D}$: $q@\{3, 39..44\} \land q.lev < \mathsf{lev}(i)$,
$\mathcal{E}$: $q@\{4..8\} \land q.break\_level < \mathsf{lev}(i)$,
$\mathcal{F}$: $q@\{9..20\} \land q.lev < \mathsf{lev}(i)$, and
$\mathcal{G}$: $q@\{9, 10\} \land q.lev = \mathsf{lev}(i)$.

First, if $\mathcal{D}$ holds, $q.lev$ may only decrease while $q@\{3, 39..44\}$ holds. Moreover, by (I62), $q.break\_level = 0$ holds while $\mathcal{D}$ holds. Hence, when $q@\{4..8\}$ is established, either $q.break\_level = 0$ is preserved, or q assigns $q.break\_level := q.lev$ by executing line 3d. In either case, $\mathcal{E}$ is established.

Second, assume that $\mathcal{E}$ holds. Since $\mathsf{lev}(i) = l$ (by (F.9)), by our inductive assumption, (L4) holds for 'l' $\leftarrow q.break\_level$. Thus, in this case, q eventually executes statement 8 and establishes either $\mathcal{F}$ or $\mathcal{G}$. (Note that, since node i has a child node j, we have $\mathsf{lev}(i) < \texttt{MAX\_LEVEL}$. Therefore, statement $8.q$ cannot establish $q@\{21\}$.)

Third, if $\mathcal{F}$ holds, then by Lemma F.2, q eventually establishes $\mathcal{G}$.

It follows that, in all cases, $\mathcal{G}$ is eventually established. By (I58), $\mathcal{B}$ implies $i = \mathsf{Node}(q, \mathsf{lev}(i))$. Thus, by (I53), $\mathcal{G}$ implies $q.n = i$. Hence, if executed when $\mathcal{B} \land \mathcal{G}$ holds, statement $9.q$ establishes $q@\{10\}$, and statement $10.q$ falsifies $\mathcal{B}$. $\Box$

It follows that, in all cases, $\mathcal{C}$ is eventually established. Hence, q eventually executes statement $11[i].q$, which falsifies (P10). $\Box$

Claim 6: (P11) *leads-to* (P12).

Proof of Claim: If (P11) holds, then eventually a process q executes either $41[i].q$ or $43[i].q$. Before the execution of $41[i].q$, by (F.19) and (I14), $WaiterLock[i] \neq \bot$ holds. Hence, statement $41[i].q$ establishes $q@\{43\} \wedge q.n = i$. Thus, in either case, q eventually executes statement $43[i].q$. After its execution, we have (P12). $\Box$

Claim 7: (P12) *leads-to* (P13).

Proof of Claim: The proof of Claim 7 is similar to that of Claim 5. If (P12) holds, then by (I65), there exists a process q satisfying one of the following.

$\mathcal{A}$: $q@\{44\} \wedge q.n = i$,
$\mathcal{B}$: $Winner[i][1] = q$, and
$\mathcal{C}$: $q@\{17\} \wedge q.n = i$.

We claim that q eventually executes $17[i].q$, thus establishing (P13). First, if $\mathcal{A}$ holds, then eventually q executes statement $44[i]$, establishing $\mathcal{B}$.

Second, assume that $\mathcal{B}$ holds. We establish the following claims.

Claim 7-1: $\mathcal{B}$ *unless* $\mathcal{C}$.

Proof of Claim: The only statements that may falsify $\mathcal{B}$ are $16[i].r$ and $44[i].r$, where r is any arbitrary process.

By (I7), $Winner[i][1] = r$ holds before the execution of $16[i].r$. Thus, statement $16[i].r$ can falsify $\mathcal{B}$ only if $r = q$, in which case it establishes $\mathcal{C}$.

By (I10), $Winner[i][1] = \bot$ holds before the execution of $44[i].r$. Thus, statement $44[i].r$ cannot falsify $\mathcal{B}$. $\Box$

Claim 7-2: $\mathcal{B}$ *leads-to* $\mathcal{C}$.

Proof of Claim: Due to Claim 7-1, it suffices to show that $\mathcal{B}$ is eventually falsified. By (I33), $\mathcal{B}$ implies that one of the following holds, for some process q.

$\mathcal{D}$: $q@\{3, 39..44\} \wedge q.lev < \mathsf{lev}(i)$,
$\mathcal{E}$: $q@\{4..8\} \wedge q.break\_level < \mathsf{lev}(i)$,

$\mathcal{F}$: $q@\{9..20\} \ \wedge \ q.lev < \mathsf{lev}(i)$, and

$\mathcal{G}$: $q@\{9, 15, 16\} \ \wedge \ q.lev = \mathsf{lev}(i)$.

As shown in the proof of Claim 5-2, if one of $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$ holds, then $\mathcal{G}$ is eventually established. By (I34) and (I58), $\mathcal{B}$ implies $Winner[i][0] \neq q \ \wedge \ i = \mathsf{Node}(q, \mathsf{lev}(i))$. Thus, by (I53), $\mathcal{G}$ implies $q.n = i$. Hence, if executed when $\mathcal{B} \ \wedge \ \mathcal{G}$ holds, statement $9.q$ establishes $q@\{15\}$, statement $15.q$ establishes $q@\{16\}$, and statement $16.q$ falsifies $\mathcal{B}$. $\square$

It follows that, in all cases, $\mathcal{C}$ is eventually established. Hence, q eventually executes statement $17[i].q$, which establishes (P13). $\square$

By (F.19) and (I14), $Waiter[i] = p \ \wedge \ WaiterLock[i] \neq \perp$ holds at state t and all subsequent states. Hence, it is straightforward to prove (P13) *leads-to* (P14). Taken together with Claims 5–7, it follows that (P14) is eventually established. Therefore, some process q eventually executes either $13.q$ and $20.q$ while $q.proc = p$ holds, establishing $p \in WaitingQueue$. However, this contradicts (F.12).

Case 3: $l > 1 \ \wedge \ p.result \neq$ PRIMARY_WAITER. In this case, clearly p is a secondary waiter (*i.e.*, $p.result =$ SECONDARY_WAITER). Hence, by (I44), (F.9), Claim 1, and Claim 2, $WaiterLock[i] \neq \perp$ holds at state t and all subsequent states.

Therefore, by (I28), there exists a process q satisfying one of the following. (The following reasoning is similar to the proofs of Claims 5 and 7.)

$\mathcal{A}$: $q@\{42\} \ \wedge \ q.n = i$,

$\mathcal{B}$: $Waiter[i] = q$, and

$\mathcal{C}$: $q@\{24\} \ \wedge \ q.n = i$.

We claim that q eventually executes $24[i].q$, thus establishing $q@\{25\} \ \wedge \ q.n = i$. This in turn implies D_1, which contradicts Claim 2.

First, if $\mathcal{A}$ holds, then eventually q executes statement $42[i]$, establishing $\mathcal{B}$.

Second, assume that $\mathcal{B}$ holds. We establish the following claims.

Claim 8: $\mathcal{B}$ *unless* $\mathcal{C}$.

Proof of Claim: The only statements that may falsify $\mathcal{B}$ are $23[i].r$ and $42[i].r$, where r is any arbitrary process.

By (I8), $\mathit{Waiter}[i] = r$ holds before the execution of $23[i].r$. Thus, statement $23[i].r$ can falsify $\mathcal{B}$ only if $r = q$, in which case it establishes $\mathcal{C}$.

By (I11), $\mathit{Waiter}[i] = \perp$ holds before the execution of $42[i].r$. Thus, statement $42[i].r$ cannot falsify $\mathcal{B}$. □

Claim 9: $\mathcal{B}$ *leads-to* $\mathcal{C}$.

Proof of Claim: Due to Claim 8, it suffices to show that $\mathcal{B}$ is eventually falsified. By (I35), $\mathcal{B}$ implies

$$q@\{4..23\} \wedge q.\mathit{break\_level} = \mathsf{lev}(i) \wedge q.\mathit{side} = 2 \wedge q@\{22, 23\} \Rightarrow q.n = i. \tag{F.22}$$

Note that $\mathcal{B}$ may be established only by statement $42.q$, which also establishes $q.\mathit{result} = \texttt{PRIMARY\_WAITER}$. Thus, by using Case 2 above (which is already proven) with 'p' $\leftarrow q$, it follows that q eventually establishes $q@\{5..23\}$. Hence, by Lemma F.2, q eventually establishes $q@\{22, 23\}$.

If $\mathcal{B}$ continues to hold, then statement $22.q$ establishes $q@\{23\}$. By (F.22), if $\mathcal{B}$ continues to hold, then statement $23.q$ falsifies $\mathcal{B}$. □

It follows that, in all cases, $\mathcal{C}$ is eventually established. Hence, q eventually executes statement $24[i].q$, which establishes $q@\{25\} \wedge q.n = i$. However, this in turn implies $D_1(p, i, j, s)$, which contradicts Claim 2. Thus, we have reached the desired contradiction. □

We now prove (L4).

$$p@\{4\} \wedge p.\mathit{side} = 2 \wedge p.\mathit{break\_level} = l \quad \textit{leads-to} \quad p@\{5\} \tag{L4}$$

Proof: If the left-hand side of (L4) holds, then by (L3), eventually the right-hand side of (L3) holds. If $p@\{5\}$, then (L4) holds. Otherwise, we have either $\mathit{Spin}[p] = \mathit{true}$ or $p \in \mathit{WaitingQueue}$. By (L2), if $p@\{4\} \wedge p \in \mathit{WaitingQueue}$ holds, then eventually either $p@\{5\}$ or $\mathit{Spin}[p] = \mathit{true}$ holds. Thus, it suffices to assume $\mathit{Spin}[p] = \mathit{true}$.

If $\mathit{Spin}[p] = \mathit{true}$ holds, then it cannot be falsified while $p@\{4\}$ holds. Thus, p eventually executes statement 4, thus establishing $p@\{5\}$. □

Finally, by combining Lemma F.2 with (L4), it follows that Algorithm T is starvation-free.

BIBLIOGRAPHY

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory. Unpublished manuscript, 1999.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103. ACM, May 1999.

[3] Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89. ACM, July 2000.

[4] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–112. ACM, May 1999.

[5] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 262–272. IEEE, October 1999.

[6] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[7] R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 800–809. ACM, May 1994.

[8] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21. IEEE, 1992.

[9] R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation*, 126(1):62–73, April 1996.

[10] R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.

[11] J. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, May 1993.

[12] J. Anderson and M. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28(2):99–103, June 1988.

[13] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355. IEEE, December 1998.

[14] J. Anderson and Y.-J. Kim. Fast *and* scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, September 1999.

[15] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43. Lecture Notes in Computer Science 1914, Springer-Verlag, October 2000.

[16] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, January 2001.

[17] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2002.

[18] J. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 3–12. ACM, July 2002.

[19] J. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and-ϕ primitives. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 538–547. IEEE, May 2003.

[20] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2–3):75–110, September 2003.

[21] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM, August 1995.

[22] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[23] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[24] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100. ACM, July 2000.

[25] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 277–286. ACM, July 1998.

[26] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report CS0956, Faculty of Computer Science, Technion, Haifa, 1999.

[27] H. Buhrman, J. Garay, J. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 194–203. ACM, August 1995.

[28] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

[29] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[30] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[31] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[32] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 148–156, December 1993.

[33] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, June 1995.

[34] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[35] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.

[36] S. Fu and N.-F. Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):628–639, June 1997.

[37] P. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *Proceedings of the Sixth Symposium on Parallel Algorithms and Architectures*, pages 236–247. ACM, June 1994.

[38] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[39] S. Haldar and P. Subramanian. Space-optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable. In *Proceedings of the Eighth International Workshop on Distributed Algorithms*, pages 116–129. Lecture Notes in Computer Science 857, Springer-Verlag, 1994.

[40] S. Haldar and K. Vidyasakar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, 1995.

[41] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[42] T.-L. Huang. Fast and fair mutual exclusion for shared memory systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 224–231, June 1999.

[43] T.-L. Huang and C.-H. Shann. A comment on "A circular list-based mutual exclusion scheme for large shared-memory multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, 9(4):415–416, April 1998.

[44] T. Johnson and K. Harathi. A prioritized multiprocessor spin lock. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):926–933, September 1997.

[45] J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.

[46] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15. Lecture Notes in Computer Science 2180, Springer-Verlag, October 2001.

[47] Y.-J. Kim and J. Anderson. A space- and time-efficient local-spin spin lock. *Information Processing Letters*, 84(1):47–55, September 2002.

[48] Y.-J. Kim and J. Anderson. Timing-based mutual exclusion with local spinning. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 30–44. Lecture Notes in Computer Science 2848, Springer-Verlag, October 2003.

[49] D. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[50] L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[51] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[52] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[53] L. Lamport. The mutual exclusion problem: Part I - A theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[54] L. Lamport. The mutual exclusion problem: Part II - Statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[55] L. Lamport. On interprocess communication: Part II - Algorithms. *Distributed Computing*, 1:86–101, 1986.

[56] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[57] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness: The ethics of concurrent termination. In *Proceedings of the 8th ICALP*. Lecture Notes in Computer Science, Vol. 115, Springer Verlag, July 1981.

[58] N. Lynch and N. Shavit. Timing based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 2–11. IEEE, December 1992.

[59] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 26–29, April 1994.

[60] E. Markatos. Multiprocessor synchronization primitives with priorities. In *Proceedings of the 1991 IFAC Workshop on Real-Time Programming*, pages 1–7. Pergamon Press, 1991.

[61] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[62] J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113. ACM, April 1991.

[63] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[64] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 164–178. Lecture Notes in Computer Science 1914, Springer-Verlag, October 2000.

[65] M. Michael and M. Scott. Fast mutual exclusion, even with contention. Technical Report TR-460, University of Rochester, Rochester, NY, 1993.

[66] M. Moir and J. Anderson. Fast, long-lived renaming. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 141–155, September 1994.

[67] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[68] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/software Interface.* Morgan Kaufmann Publishers, 2nd edition, 1997.

[69] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[70] G. Peterson and J. Burns. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual ACM Symposium on Foundation of Computer Science*. ACM, 1987.

[71] G. Pfister and V. Norton. “Hot spot” contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[72] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, Cambridge, Massachusetts, 1986.

[73] I. Rhee and C.-Y. Lee. An efficient recovery-based spin lock protocol for preemptive shared-memory, multiprocessors. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 77–86, May 1996.

[74] R. Schaffer. On the correctness of atomic multi-writer registers. Technical Report MIT/LCS/TM-364, Laboratory for Computer Science, MIT, Cambridge, 1988.

[75] M. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21th Annual ACM Symposium on Principles of Distributed Computing*, pages 31–40. ACM, July 2002.

[76] M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the Eighth Annual ACM Symposium on Principles and Practice of Parallel Programming*, pages 44–52. ACM, June 2001.

[77] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, 1994.

[78] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. ACM, August 1992.

[79] E. Styer and G. Peterson. Tight bounds for shared memory symmetric mutual exclusion. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–191. ACM, August 1989.

[80] Y.-K. Tsay. Deriving a scalable algorithm for mutual exclusion. In *Proceedings of the 12th International Symposium on Distributed Computing*, pages 393–407. Springer Verlag, September 1998.

[81] P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

[82] R. Wisniewski, L. Kontothanassis, and M. Scott. Scalable spin locks for multiprogrammed systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 26–29, April 1994.

[83] R. Wisniewski, L. Kontothanassis, and M. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practices of Parallel Programming*, pages 199–206. ACM, July 1995.

[84] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

[85] X. Zhang, Y. Yan, and R. Castañeda. Evaluating and designing software mutual exclusion algorithms on shared-memory multiprocessors. *IEEE Parallel and Distributed Technology*, pages 25–42, Spring Issue, 1996.