Nonatomic Mutual Exclusion with Local Spinning*

Yong-Jik Kim
Tmax Soft Research Center
272-6 Seohyeon-dong, Seongnam-si
Gyeonggi-do, Korea 463-824
Email: jick@tmax.co.kr

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
Email: anderson@cs.unc.edu

April 2003, Revised October 2005

Abstract

We present an N-process local-spin mutual exclusion algorithm, based on nonatomic reads and writes, in which each process performs $\Theta(\log N)$ remote memory references to enter and exit its critical section. This algorithm is derived from Yang and Anderson's atomic tree-based local-spin algorithm in a way that preserves its time complexity. No atomic read/write algorithm with better asymptotic worst-case time complexity (under the remote-memory-references 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-path algorithms* (in which contention-free time complexity is required to be $O(1)$) or *adaptive algorithms* (in which time complexity is required to depend only on the number of contending processes). 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.

1 Introduction

This paper is concerned with shared-memory mutual exclusion algorithms based on read and write operations.[1] In work on such algorithms, *nonatomic* and *local-spin* algorithms have received considerable attention. In nonatomic algorithms, variable accesses 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 [22]. Thus, mutual exclusion algorithms requiring this are in some sense circular.

In local-spin algorithms, all busy-waiting loops are read-only loops in which only locally-accessible variables are read; a variable is locally accessible if it is in a local cache line (possible on a multiprocessor with coherent caches) or stored in a local memory partition (possible on a distributed shared-memory machine). By structuring busy-waiting loops in this way, contention for the processors-to-memory interconnection network can be greatly reduced. Performance studies presented in several papers [13, 17, 27, 34] have shown that local-spin algorithms typically scale well as contention increases, while non-local-spin algorithms do not.

In this paper, we present possibility and impossibility results concerning the time complexity of mutual exclusion algorithms that are *both* nonatomic *and* use local spinning. We exclusively use the *RMR* (*remote-memory-reference*) *measure* to assess time complexity. As its name suggests, only remote memory references that cause an interconnect traversal are counted under this measure. We assess the RMR time complexity

*Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0208289.
[1] All claims made hereafter are assumed to pertain to this class of algorithms unless otherwise indicated.

of an algorithm by counting the total number of remote memory references required by one process to enter and then exit its critical section once.

Before describing our main contributions, we first give a brief overview of relevant related research on nonatomic and local-spin algorithms.

Nonatomic algorithms. Lamport was the first to point out the circularity inherent in assuming atomic statement execution [22]. He also presented the first nonatomic algorithm, his famous bakery algorithm [22]. Lamport's work on the bakery algorithm was a catalyst for much subsequent work by him on proof formalisms for nonatomic algorithms (*e.g.*, [26]).

The bakery algorithm is not a local-spin algorithm. In addition, it requires unbounded memory. In later work, Lamport presented four other nonatomic algorithms, each with bounded memory [23]. These algorithms differ in the progress and fault-tolerance properties they satisfy. None are local-spin algorithms.

Local-spin algorithms. The earliest local-spin algorithm based only on reads and writes is also the only prior *nonatomic* local-spin algorithm known to us [6]. This algorithm, due to Anderson, 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 [22, 24]. 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 [22]. 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 re-write a variable with its prior value have no adverse impact.

In later work, Yang and Anderson showed that sub-linear time complexity was possible, at the price of atomic memory. They established this by presenting an algorithm with $\Theta(\log N)$ RMR time complexity, in which instances of an $O(1)$ two-process algorithm are embedded in a binary arbitration tree [34]. Their two-process algorithm, unlike Anderson's, does not require *statically* allocated single-writer, single-reader variables. Using such variables in an arbitration tree is problematic because the participating processes at each node may vary with time. On the other hand, Yang and Anderson's algorithm uses multi-bit variables, and hence does not work if variable accesses are nonatomic.

Fast and adaptive algorithms. In recent years, there has been much interest in algorithms that are *fast*, *i.e.*, those that take $O(1)$ steps in the absence of contention [9, 25, 34] or that are *adaptive*, *i.e.*, with time complexity (under some measure) that is a function of the number of contending processes, independent of the total number or processes in the system [3, 4, 8, 14, 16, 31]. In a recent paper [9], 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 another recent paper [8], we 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," that is, the maximum number of processes that are active simultaneously [1]. A similar algorithm, developed independently, was recently published by Afek *et al.* [4].

Lower bounds. Several prior research efforts on lower bounds are of relevance to this paper. Of particular relevance are lower bounds established by Anderson and Yang, which involve trade-offs between time complexity and write-contention [12]. The *write-contention* (*access-contention*) of a concurrent program is

[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.

the number of processes that may potentially be simultaneously enabled to write (access) 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. 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 [5], Alur and Taubenfeld 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).

In other related work [11], 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 a later related paper [21], we proved the impossibility of an adaptive *atomic* algorithm with $o(k)$ RMR time complexity, where k is point contention.)

Contributions. 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 paper. 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, fast and adaptive algorithms are impossible even if caching techniques are used to avoid accessing the interconnection network. Some of the techniques used to establish these bounds are taken from earlier papers [11, 12], while others are new, being applicable when memory accesses are nonatomic. Given the prior results summarized above, it follows that any fast or adaptive algorithm necessarily must use some $\Omega(\log N)$-bit variables, some $\Omega(N^\epsilon)$-writer variables, *and* some atomic variables.

Organization. The rest of this paper is organized as follows. In Section 2, our nonatomic algorithm is presented; a correctness proof for the algorithm is given in an appendix. Definitions needed to establish the above-mentioned lower bound are then given in Section 3. The lower-bound proof is sketched in Section 4; a full proof is given in a second appendix. We conclude in Section 5.

2 Nonatomic Algorithm

As mentioned earlier, our nonatomic algorithm is derived from Yang and Anderson's algorithm. We hereafter denote these two algorithms as Algorithms NA and YA, respectively; both are depicted in Figure 1. In this figure, "**await** B" is used as a shorthand for "**while** $\neg B$ **do** /* null */ **od**," where B is a boolean expression.

Algorithm YA. We begin with a brief, informal description of Algorithm YA [34]. Associated with each node n at height h in the arbitration tree is a two-process mutual exclusion algorithm, which uses the following variables: $C[n][0]$, $C[n][1]$, $T[n]$, and a subset of $P[h][0], \ldots, P[h][N-1]$. Variable $C[n][0]$ ranges

Algorithm YA (The original algorithm in [34])

```
process p ::   /* 0 ≤ 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, ⊥)
        initially ⊥;
    P: 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;

2:  for h := 1 to L do
3:      node := ⌊(N + p)/2^h⌋;
4:      side := ⌊(N + p)/2^(h−1)⌋ mod 2;
5:      C[node][side] := p;
6:      T[node] := p;
7:      P[h][p] := 0;
8:      rival := C[node][1 − side];
9:      if (rival ≠ ⊥ ∧ T[node] = p) then
10:         if P[h][rival] = 0 then
11:             P[h][rival] := 1 fi;
12:         await P[h][p] ≥ 1;
13:         if T[node] = p then
14:             await P[h][p] = 2 fi
        fi
    od;

15: Critical Section;

16: for h := L downto 1 do
17:     node := ⌊(N + p)/2^h⌋;
18:     side := ⌊(N + p)/2^(h−1)⌋ mod 2;
19:     C[node][side] := ⊥;
20:     rival := T[node];
21:     if rival ≠ p then
22:         P[h][rival] := 2 fi
    od
od
```

(a)

Algorithm NA (New nonatomic algorithm)

```
process p ::   /* 0 ≤ p < N */

/* all variable declarations are as in */
/* ALGORITHM YA, 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;

2:  for h := 1 to L do
3:      node := ⌊(N + p)/2^h⌋;
4:      side := ⌊(N + p)/2^(h−1)⌋ mod 2;
5:      C[node][side] := p;
6:      T[node] := p;
7:      rtoggle := ¬R1[h][p];
8:      R2[h][p] := rtoggle;
9:      qtoggle := ¬Q1[h][p];
10:     Q2[h][p] := qtoggle;
11:     rival := C[node][1 − side];
12:     if (rival ≠ ⊥ ∧ T[node] = p) then
13:         temp := Q2[h][rival];
14:         Q1[h][rival] := temp;
15:         await (Q1[h][p] = qtoggle) ∨
16:               (R1[h][p] = rtoggle);
17:         if T[node] = p then
18:             await R1[h][p] = rtoggle fi
        fi
    od;

19: Critical Section;

20: for h := L downto 1 do
21:     node := ⌊(N + p)/2^h⌋;
22:     side := ⌊(N + p)/2^(h−1)⌋ mod 2;
23:     C[node][side] := ⊥;
24:     rival := T[node];
25:     if rival ≠ p then
26:         temp := R2[h][rival];
27:         R1[h][rival] := temp fi
    od
od
```

(b)

Figure 1: **(a)** Algorithm YA and **(b)** its nonatomic variant. In (b), reads and writes of the C and T variables are assumed to be implemented using register constructions.

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 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. Ties are broken in favor of the first process to update $T[n]$. Variable $P[h][p]$ 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).

Loosely speaking, the two-process algorithm at node n works as follows. A process l from the left subtree rooted at n "announces" its arrival at node n by establishing $C[n][0] = l$. It then assigns its identifier l to the tie-breaker variable $T[n]$, and initializes its spin variable $P[h][l]$. If no process from the right-side has attempted to acquire node n, *i.e.*, if $C[n][1] = \perp$ holds when l executes line 8, then process l proceeds directly to the next level of the arbitration tree (or to its critical section if n is the root). Otherwise, if $C[n][1] = r$, where r is some right-side process, then l reads the tie-breaker variable $T[n]$. If $T[n] \neq l$, then process r has updated $T[n]$ *after* process l, so l can enter its critical section (recall that ties are broken in favor of the first process to update $T[n]$). If $T[n] = l$ holds, then either process r executed line 6 before process l did, or process r has executed line 5 but not line 6. In the first case, l should wait until r "releases" node n in its exit section, whereas, in the second case, l should be able to proceed past node n. This ambiguity is resolved by having process l execute lines 10–14. Lines 10–11 are executed by process l to release process r in the event that it is waiting for l to update the tie-breaker variable (*i.e.*, r is busy-waiting at node n at line 12). Lines 12–14 are executed by l to determine which process updated the tie-breaker variable first. Note that $P[h][l] \geq 1$ implies that r has already updated the tie-breaker, and $P[h][l] = 2$ implies that r has released node n. To handle these two cases, process l first waits until $P[h][l] \geq 1$ holds (*i.e.*, until r has updated the tie-breaker), re-examines $T[n]$ to see which process updated it last, and finally, if necessary, waits until $P[h][l] = 2$ holds (*i.e.*, until process r releases node n).

After executing its critical section, process l releases node n by establishing $C[n][0] = 0$. If $T[n] = r$, in which case process r is competing at node n, then process l updates $P[h][r]$ so that process r does not block at node n.

To see that Algorithm YA is a local-spin algorithm, note that each process only waits on spin variables dedicated to it. On a distributed shared-memory machine, a process's spin variables can be stored in a local memory partition. On a cache-coherent machine, the first read of a spin variable by a process p in a busy-waiting loop creates a cached copy. All subsequent reads by p until the variable is written by another process are handled in-cache. The algorithm ensures that after such a write, p's busy-waiting loop terminates.

Algorithm NA. In the rest of this section, we consider the problem of converting Algorithm YA into a nonatomic algorithm. The notion of a nonatomic variable that we assume is that captured by Lamport's definition of a *safe register* [24]: 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 into a nonatomic algorithm is to implement each atomic variable using nonatomic ones by applying wait-free register constructions presented previously [18, 19, 24, 28, 29, 30]. This is in fact the approach we take for the C and T variables. However, if such constructions are applied to implement the P 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 [24].) As a result, the spins in lines 12 and 14 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 [18]. 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 [19, 24, 28, 29, 30]. For variable $T[n]$, the number of readers and writers is constant. 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 [20]. (For ease of exposition, this is not done in Algorithm NA in Figure 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 [6].

The P variables can be dealt with similarly. In Algorithm YA, the condition $P[h][p] \geq 1$ indicates that process p may proceed past its first **await**, and the condition $P[h][p] = 2$ 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 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 lines 7–10 and waits while this condition continues to hold at lines 15–16 (note that $qtoggle = Q2[h][p] \wedge rtoggle = R2[h][p]$ holds while p continues to wait).[3] A rival process at node n seeks to establish $Q1[h][p] = Q2[h][p]$ at lines 13–14; the effect is similar to lines 10–11 in Algorithm YA. Lines 18 and 26–27 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.[4] A complete correctness proof for the algorithm is given in Appendix A. This gives us the following theorem.

Theorem 1 *The mutual exclusion problem can be solved with $\Theta(\log N)$ RMR time complexity using only nonatomic reads and writes.* □

3 Lower Bound: System Model

In this section, we present the model of a nonatomic shared-memory system that is used in our lower-bound proof. This model is similar to that used in [11, 12].

Shared-memory systems. A *shared-memory system* $\mathcal{S} = (C, P, V)$ consists of a set of computations C, a set of processes P, and a set of shared variables V. A *computation* is a finite sequence of events. To complete the definition of a shared-memory system, we must 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 section. As needed terms are defined, various notational conventions are also introduced that will be used in the rest of the paper.

Informally, an *event* is a particular execution of a statement of some process that involves either reading or writing a shared variable. (However, a nonatomic write is actually represented by two events, one for its beginning and one for its end; see below.) 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.

[3] In the preliminary version of this paper [10], lines 9–10 in Figure 1(b) precede lines 7–8, writing $Q2$ before $R2$. We found that that version is susceptible to livelock.

[4] In fact, because $Q2$ and $R2$ are not accessed within any busy-waiting loop, they can be assumed to be atomic, as they can be implemented nonatomically via register constructions. On the other hand, $Q1$ and $R1$ *are* accessed within busy-waiting loops, so implementing them using register constructions would result in non-local spinning.

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. According to the definitions below, a read of a variable that overlaps a nonatomic write of that variable may return any value, as assumed earlier in Section 2.

We now formalize these ideas. 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.) 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.) For brevity, we sometimes use e_p to denote an event of 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*. □

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 P4, 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 2(a). If a process has performed an invocation event, then it may execute the matching response event (Property P5). As stated before, our proof strategy 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 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 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 P3 below). For example, in Figure 2(a), events c–f may read any value from v.

Notational conventions pertaining to computations. The value of variable v at the end of computation H, denoted $value(v, H)$, 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 $last_writer_event(v, H)$,[6] and the process that executes that event is denoted $last_writer(v, H)$. More formally, let e_p be the last event in H with operation $\mathsf{write}(v)$, $\mathsf{invoke}(v)$, or $\mathsf{respond}(v)$. We define $last_writer_event(v, H) = e_p$, $last_writer(v, H) = p$, and $value(v, H)$ to be the value written to v by e_p. If e_p is an invocation event $[p, \mathsf{invoke}(v), \alpha]$, then we define $value(v, H)$ to

[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.

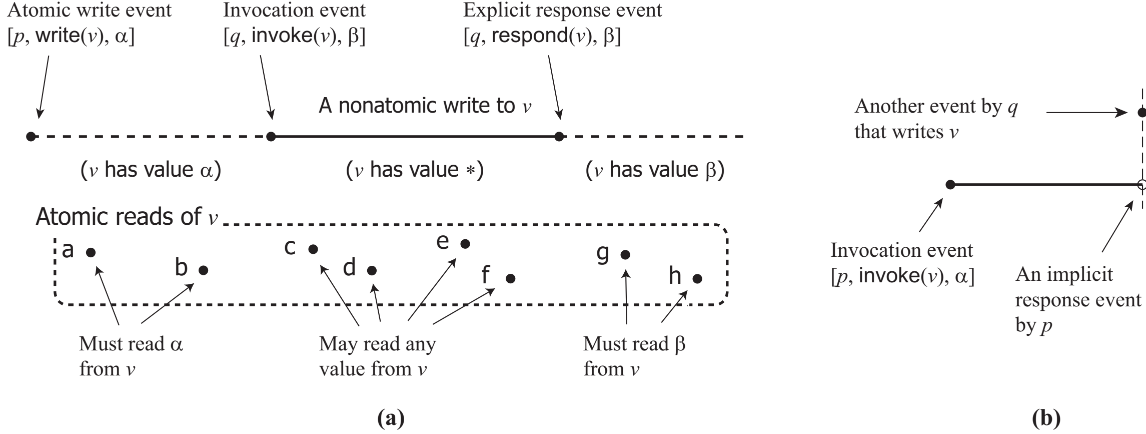


Figure 2: Overlapping reads and writes of the same variable. **(a)** A nonatomic write to v by p, terminated by an explicit response event. **(b)** A nonatomic write to v by p, terminated by another process q's write to v. In this case, q's event may be an atomic write event or an invocation event.

be $\star$, not α. If v is not written by any event in H, then we define $last_writer(v, H) = \bot$ and $last_writer_event(v, H) = \bot$.

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, $\langle\rangle$ to denote the empty computation, and $H \circ G$ to denote the computation obtained by concatenating computations H and G. For a computation H and a set of processes Y, $H \,|\, Y$ denotes the subcomputation of H that contains all events in H of processes in Y.[7] If G is a subcomputation of H, then $H - G$ is the computation obtained by removing all events in G from H. A computation H is a *Y-computation* if and only if $H = H \,|\, Y$. For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes (*e.g.*, $H \,|\, p$ instead of $H \,|\, \{p\}$).

Properties of shared-memory systems. The following properties apply to any shared-memory system. (Note that these properties are not complete, *i.e.*, they do not generate the complete set of all valid computations. These "incomplete" rules are sufficient for our purpose.)

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 $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \,|\, p = H \,|\, p$. Also assume that either **(i)** e_p is not a read event, or **(ii)** e_p reads v and $value(v, G) \in \{value(v, H), \star\}$ holds. Then, $G \circ \langle e_p \rangle \in C$ holds.

— *Informally, assume that two computations H and G are not distinguishable to process p, and that p can execute an event e_p after H. If e_p is not a read event, then it can be executed also after G. On the other hand, if e_p is a read event of some variable v, then it can be executed after G if v's value after G is either $\star$ or the same value as after H. 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.*

P3: For any $H \in C$ and a read event $e_p = [p, \mathsf{read}(v), \alpha]$, $H \circ \langle e_p \rangle \in C$ implies that $value(v, H)$ 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.*

[6] 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.)

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

P4: 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 $[p, \mathsf{invoke}(v), \alpha]$).*

P5: 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 $\mathit{last_writer_event}(v, H) = [p, \mathsf{invoke}(v), \alpha]$ 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, we hereafter assume that every computation of concern is free of explicit response events.

One-shot mutual exclusion systems. We now define a special kind of shared-memory system, namely 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.

A *one-shot mutual exclusion system* $\mathcal{S} = (C, P, V)$ is a 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: $\mathit{Enter}_p = [p, \mathsf{write}(\mathit{entry}_p), 0]$, and $\mathit{CS}_p = [p, \mathsf{write}(\mathit{cs}_p), 0]$. (Since private variables are completely ignored in our formalism, entry_p and cs_p are considered "shared" variables, even though they are accessed only by p.)

These events are allowed in the following situations. For all $H \in C$,

- if $H \,|\, p \neq \langle\rangle$, then the first event (and only the first event) by p in H is Enter_p;
- if $H \,|\, p = \langle\rangle$, then $H \circ \langle \mathit{Enter}_p \rangle \in C$;
- if H contains CS_p, then it is the last event of $H \,|\, p$.

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.)

In our proof, Enter_p is used as an easy way to construct our "first" computation, that is, a computation in which every process p executes only its first operation, Enter_p. (Without Enter_p, constructing the "first" computation would require an additional mechanism, since our proof requires that processes do not gain knowledge of each other. By introducing Enter_p, this is automatically satisfied.) 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 $H \circ G \circ \langle \mathit{CS}_p \rangle \in C$ 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. Since our lower bound is concerned with the number of distinct variables accessed, in order to facilitate the proof, we define certain events as *critical events*. 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).

write $\boldsymbol{entry_p}$ **:= 0;** /* $Enter_p$ */
write $\boldsymbol{u}$ **:= 1;**
read $\boldsymbol{v}$ **= 2;**
read $\boldsymbol{u}$ **= 1;**
read $v = 2$;
write $u := 3$;
write $\boldsymbol{w}$ **:= 4;**
write $\boldsymbol{cs_p}$ **:= 0** /* CS_p */
halt

(a)

$S(p,1)$: **write** $\boldsymbol{entry_p}$ **:= 0;** /* $Enter_p$ */
$S(p,2)$: **write** $\boldsymbol{u}$ **:= 1;**
$S(p,3)$: **read** $\boldsymbol{v}$ **= 2;**
$S(p,4)$: **read** $\boldsymbol{u}$ **= 1;** read $v = 2$; write $u := 3$;
$S(p,5)$: **write** $\boldsymbol{w}$ **:= 4;**
$S(p,6)$: **write** $\boldsymbol{cs_p}$ **:= 0** /* CS_p */
halt

(b)

Figure 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 segments.

Consider the "solo" computation S_p by a process p, such that every write in it is atomic. As defined above, the first event in S_p must be $Enter_p$. By the Progress property, p eventually executes CS_p, and then terminates. (An example is shown in Figure 3(a).) We define $ce(p, j)$ as the jth critical event by p in its solo computation. For example, in Figure 3(a), we have $ce(p, 1) = Enter_p$, $ce(p, 2) = [p, \mathsf{write}(u), 1]$, $ce(p, 3) = [p, \mathsf{read}(v), 2]$, 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 3, we have $ie(p, 2) = [p, \mathsf{invoke}(u), 1]$ and $ie(p, 5) = [p, \mathsf{invoke}(w), 4]$, 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. Note that the first (respectively, last) solo segment of p consists of a single event, $Enter_p$ (respectively, CS_p). An example is shown in Figure 3(b).

4 Lower Bound: Proof Sketch

In Appendix B, we show that for any nonatomic 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 in isolation, where $N = |P|$. In this section, we sketch the key ideas of our proof.

4.1 Brief Overview

Our proof focuses on a special class of computations called "regular" computations. 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.*, $\langle Enter_p\rangle$ 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.[8] This has three consequences: each process executes the same sequence of events as its solo computation (*i.e.*, as it does when it is executed alone); we can "erase" any active process

(*i.e.*, remove its events from the computation) and still get a valid computation; each active process has a "next" critical event, and hence, a "next" solo segment. In each induction step, we append to each of the n active processes 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. Thus, at each induction step, a process may undergo one of the following changes: **(i)** an active process is erased, **(ii)** a covering process is erased, or **(iii)** an active process becomes a covering process. (A covering process never becomes active again.) These basic techniques, covering and erasing, have been previously used to prove other lower bounds pertaining to concurrent systems [2, 11, 15, 21, 32]. However, the particular covering strategy being used here is different from those applied in earlier papers, 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 jth induction step (and are not erased so far).[9] Thus, the structure of H is as shown in Figure 4.

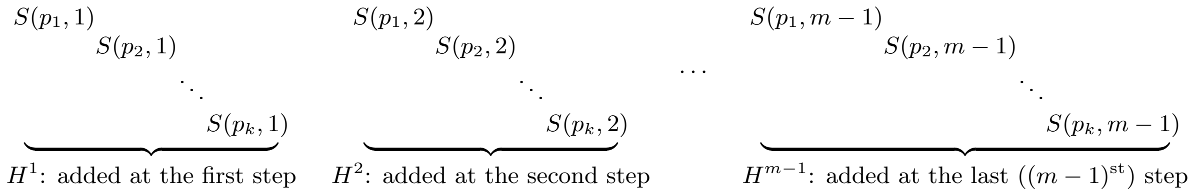


Figure 4: Structure of a regular computation H after $m-1$ induction steps. This diagram does not show the full structure of H; additional details will be introduced as needed.

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.

> **Property A:** For each variable v, a regular computation H may contain at most one process that executes "uncovered" write(s) of v.

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 it may gain knowledge of p, which is clearly undesirable. Property A 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, each active process p_j has its "next" solo segment, $S(p_j, m)$, which can be potentially executed after H. We now present examples that demonstrate why and when the two strategies — covering and erasing — are necessary.

Example of the erasing strategy. First, we consider an "ideal" case, shown in Figure 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*

[8] 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.

[9] We consider the initial computation H_1 to have taken one induction step to construct. Thus, a computation after $m-1$ induction steps has $m-1$ solo segments per each active process.

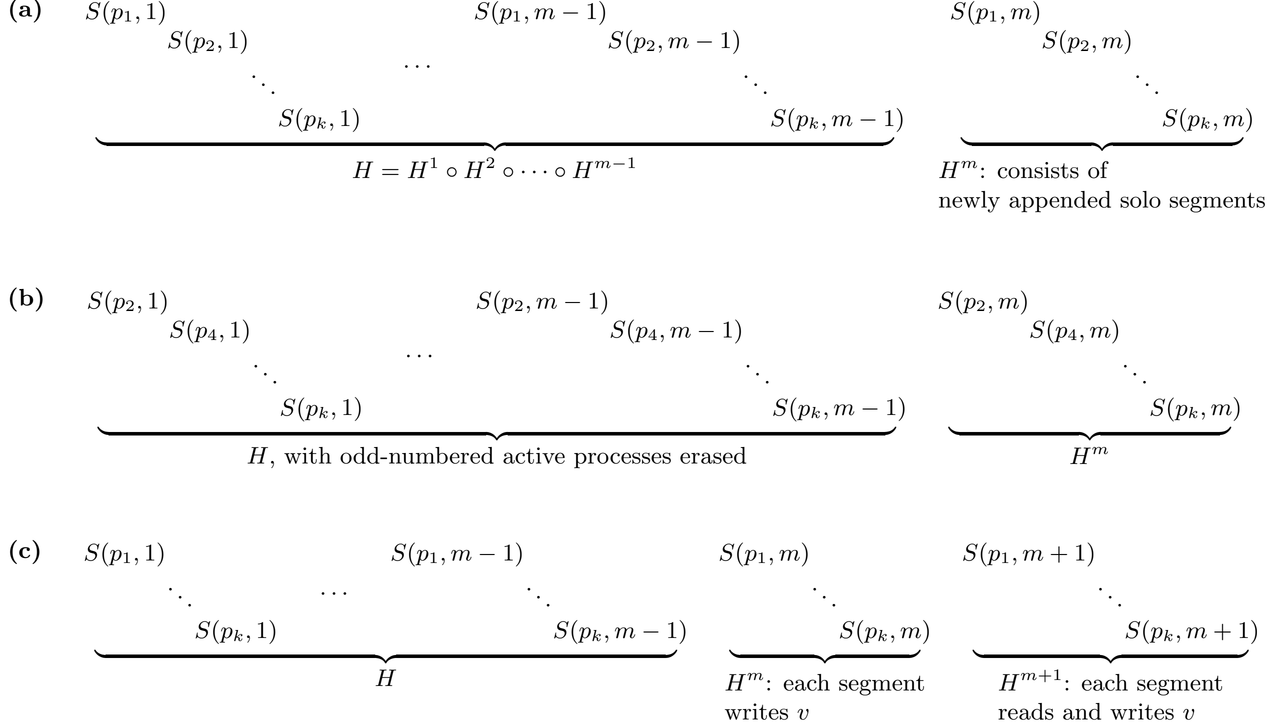


Figure 5: Extensions of H: only relevant details are shown. **(a)** An ideal case. Each next critical event $ce(p_j, m)$ accesses a distinct variable v_j, which is not accessed in H. **(b)** Erasing strategy. Each next critical event $ce(p_j, m)$ writes variable $v_{\lceil j/2 \rceil}$, which is not 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.

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 5(a).

In this ideal case, no erasing or covering is necessary. However, consider another case, depicted in Figure 5(b), in which k active processes access $k/2$ distinct variables, where for simplicity, k is assumed to be even. Suppose that each variable v_j (for $1 \le j \le k/2$) 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 A 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 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 Property A 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. Assume that next critical events collectively access only a small number of variables. 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 Property A, 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 the same variable v. (That is, $ce(p_j, m)$ is a write event of v, for each $p_j \in \mathrm{Act}(H)$.) In this case, appending all next solo segments as in Figure 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 $(m+1)^{\mathrm{st}}$ critical event $ce(p_j, m+1)$. Moreover, suppose that each $(m+1)^{\mathrm{st}}$ solo segment $S(p_j, m+1)$ contains noncritical write(s) of v. This situation is depicted in Figure 5(c).

Although simple, this is the "worst case" scenario in a sense: every process, in its $(m+1)^{\mathrm{st}}$ 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. (Similarly, each p_{j+1} reads from v the value written by p_j in $S(p_j, m+1)$.)

Information flow may be eliminated here by changing some critical writes into invocation events on the same variable. Since the $(m+1)^{\mathrm{st}}$ solo segments contain noncritical writes of v, we must include an invocation event on v after each $(m+1)^{\mathrm{st}}$ 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 odd-numbered ones) and letting the other half continue their active execution, we append the following computations at the mth and $(m+1)^{\mathrm{st}}$ steps, respectively:

$$\begin{aligned} H^m &= S(p_2, m) \circ S(p_4, m) \circ \cdots \circ S(p_k, m), \\ H^{m+1} &= \langle ie(p_1, m) \rangle \circ S(p_2, m+1) \circ \langle ie(p_3, m) \rangle \circ S(p_4, m+1) \circ \cdots \circ \\ &\qquad \langle ie(p_{k-1}, m) \rangle \circ S(p_k, m+1). \end{aligned}$$

Here, H^m consists of the solo segments of all even-numbered active processes. H^{m+1} starts with an invocation event by p_1 on v, so that the following critical read of v in $S(p_2, m+1)$ does not gain knowledge of p_k. Similarly, in H^{m+1}, solo segments by even-numbered active processes are interleaved with invocation events on v by odd-numbered processes, so that information flow among them is prevented. We thus guarantee that any read from v by a process p_j (for even 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 even-numbered ones), plus $k/2$ covering processes (the odd-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 $(m+1)^{\mathrm{st}}$), 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 $l \geq m$. Thus, segments appended at the mth and later steps may have the following structure.

$$\begin{array}{rcl} H^m & = & S(p_1, m) \circ \langle ie(p_2, m) \rangle \circ S(p_{s+2}, m) \circ \langle ie(p_{s+3}, m) \rangle \circ \cdots \\ H^{m+1} & = & S(p_1, m+1) \circ \langle ie(p_3, m) \rangle \circ S(p_{s+2}, m+1) \circ \langle ie(p_{s+4}, m) \rangle \circ \cdots \\ H^{m+2} & = & S(p_1, m+2) \circ \langle ie(p_4, m) \rangle \circ S(p_{s+2}, m+2) \circ \langle ie(p_{s+5}, m) \rangle \circ \cdots \\ \cdots & & \cdots \end{array} \tag{1}$$

Note that some of these invocation events are actually unnecessary. For example, event $ie(p_2, m)$ is unnecessary because the following segment $S(p_{s+2}, m)$ starts with a write to v (*i.e.*, $ce(p_{s+2}, m)$), 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 (noncritical) write to v. Thus, it may be overkill to insert an invocation event after every $S(p_j, l)$. We still include such unnecessary invocation events to simplify bookkeeping.

A generic description of the covering strategy. The structure depicted in (1) 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 mth critical events (by all active processes) may collectively access multiple variables. In that case, we have to apply the covering strategy separately to each variable. Finally, it is generally impossible to index and arrange processes in a regular fashion as above, since some of these active processes may be erased later.

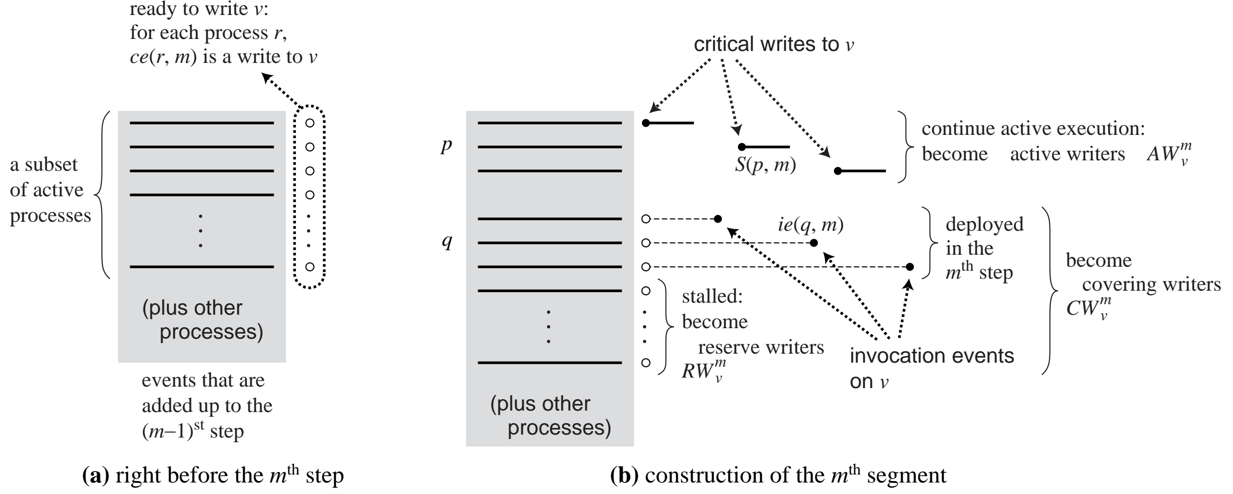
Thus, we need a more dynamic approach, as depicted in Figure 6. Assume that, at the mth induction step, we find that there are "too many" processes whose m^{th} critical event is a write to v, as shown in Figure 6(a). (In (1), these processes comprise all active processes; in general, they will form 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 (see Figure 6(b); the "reserve writers" in this figure will be considered later). For example, in (1), processes p_1 and p_{s+2} belong to AW_v^m, while processes $p_2,\ p_3,\ p_4,\ p_{s+3},\ p_{s+4}$, and p_{s+5} 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 mth solo segment $S(p, m)$ to construct the mth 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 $S(p, m)$ (see Figure 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 lth induction step, where $l > m$ (see Figure 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 lth segment. Since $ce(p, m)$ writes v, $S(p, l)$ may contain a (noncritical) 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 2 in Appendix B.)

Note that, after the construction of 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 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 mth step, depicted in Figure 6(b)) but not in H_l (depicted in Figure 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 (which is a continuation of Figure 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.



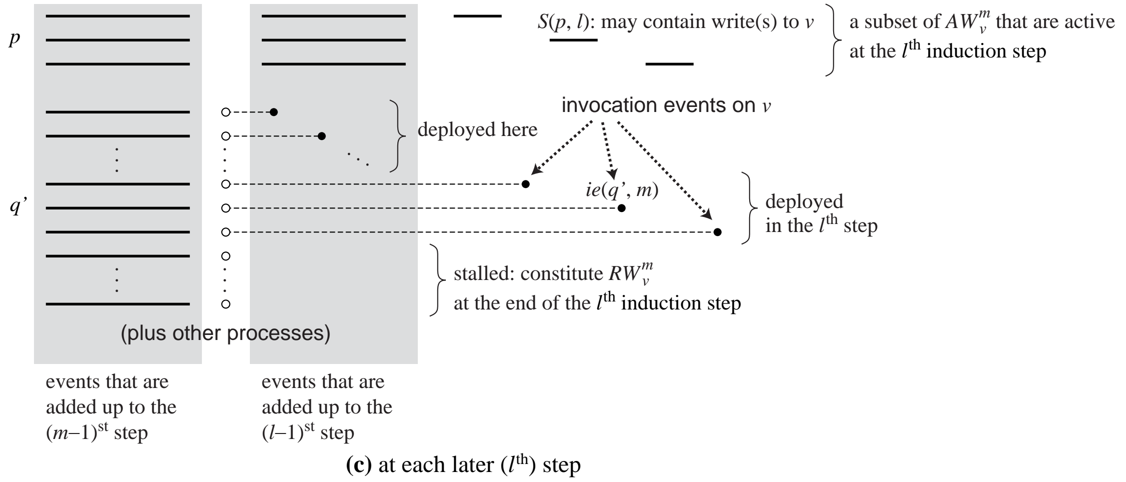


Figure 6: Covering strategy. We only show relevant processes. (In general, the computation has many other processes that are not depicted here.) Here and in later figures, horizontal lines depict events of a particular process, black circles (•) depict a single event, and empty circles (◦) depict an event that is *enabled* at that point but not executed. **(a)** At the mth 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 mth 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 lth 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 $S(p, l)$.

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).

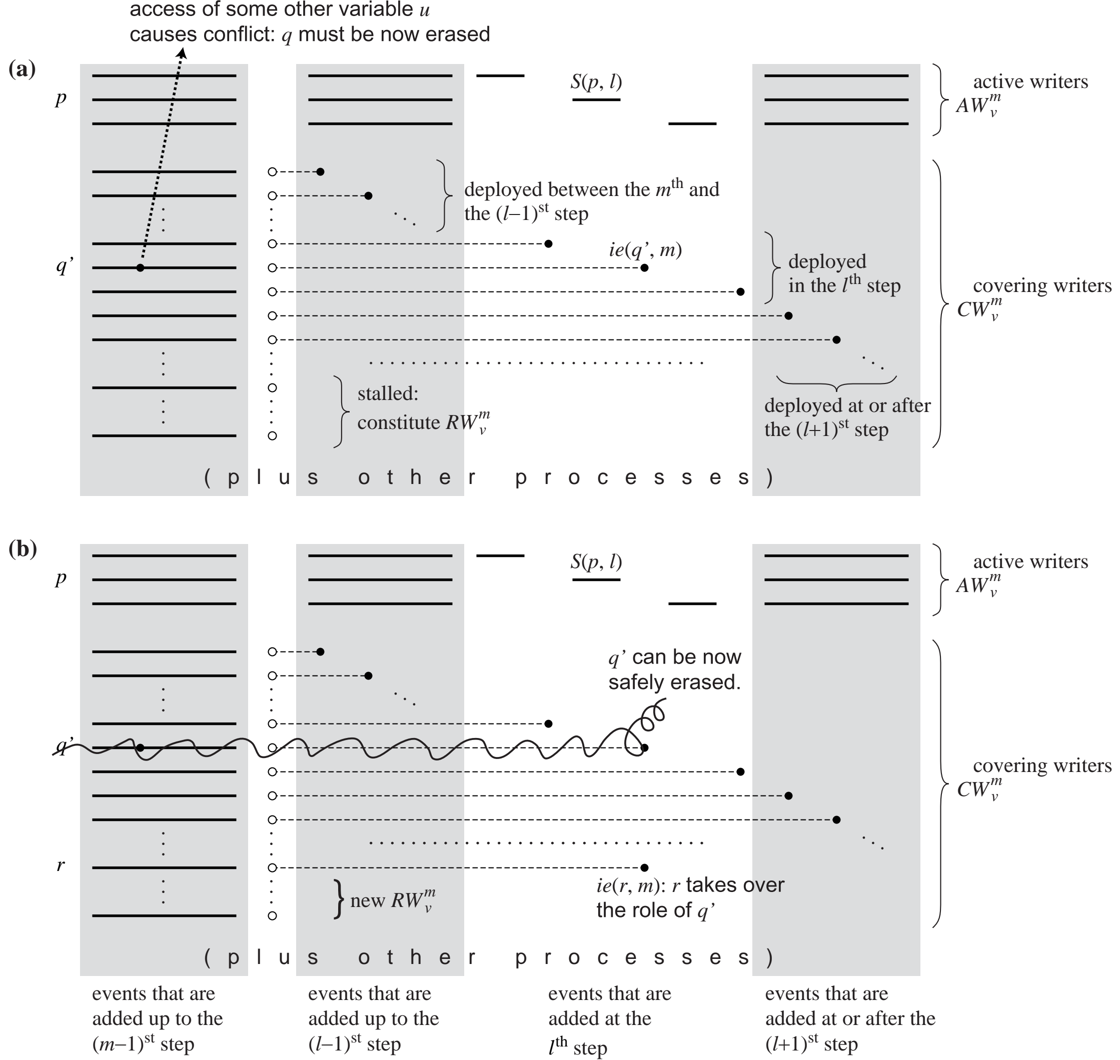


Figure 7: The use of reserve processes to "exchange" two processes before erasing. **(a)** After Figure 6(c), at some later step, we find that process $q' \in CW_v^m$ incurs a conflict via some other variable u. (For example, q' may have executed an *uncovered* write of u at some earlier step, and then we later find that, at some k^{th} step ($k > l$), *all* remaining active processes read 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.

4.2 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 mth *segment* of H. For each *segment index* m ($1 \le m \le m_H$), H^m consists of the events that are appended at the mth induction step (and are not erased so far), and contains exactly one critical event by each process that was active at the mth 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:

$$P(H) = \{p \in P\colon H \,|\, p \neq \langle\rangle\}. \tag{2}$$

Processes in $P(H)$ are partitioned into two sets: $\mathrm{Act}(H)$, the *active processes*, and $\mathrm{Cvr}(H)$, the *covering processes*.

$$P(H) = \mathrm{Act}(H) \cup \mathrm{Cvr}(H) \quad \wedge \quad \mathrm{Act}(H) \cap \mathrm{Cvr}(H) = \{\}. \tag{3}$$

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 mth induction step to cover v:

$$CW_v^m = \{p \in \mathrm{Cvr}(H)\colon (ci(p), cv(p)) = (m, v)\}. \tag{4}$$

By definition, we also have the following:

$$(ci(p), cv(p)) = (m, v) \quad \text{only if} \quad ce(p, m) \text{ is a write to } v. \tag{5}$$

As explained before, a process p in CW_v^m may be deployed to cover $S(q, l)$, for some process q in AW_v^m and segment index $l \geq m$. In this case, we define $cp(p)$, the process covered by p, to be q. We let $cp(p) = \perp$ if p is not deployed in H.

The covering processes may also be grouped as follows: we define $\mathrm{Cvr}^m(H)$, the set of covering processes *at the end of the mth segment*, as follows:

$$\mathrm{Cvr}^m(H) = \{p \in \mathrm{Cvr}(H)\colon ci(p) \leq m\} = \bigcup_{1 \leq j \leq m,\ v \in V} CW_v^j. \tag{6}$$

$\mathrm{Cvr}^m(H)$ consists of processes that are selected for covering through the mth induction step. We similarly define $\mathrm{Act}^m(H)$, the set of active processes *at the end of the mth segment*, as follows:

$$\mathrm{Act}^m(H) = P(H) - \mathrm{Cvr}^m(H). \tag{7}$$

$\mathrm{Act}^m(H)$ consists of processes that have *not* been selected for covering, and hence are active at the end of the mth induction step. A process p in $\mathrm{Act}^m(H)$ may be selected for covering in some later, say, l^{th}, induction step, in which case p belongs to both $\mathrm{Act}^m(H)$ and $\mathrm{Cvr}^l(H)$ (see Figure 8). Note that if a process q is selected to cover a variable v at the mth induction step (*i.e.*, $q \in CW_v^m$), then q does not become active again. That is, $q \in \mathrm{Cvr}^m(H)$ implies $q \notin \mathrm{Act}^{m'}(H)$, for each $m' \geq m$.

From the description above, we have $\mathrm{Cvr}(H) = \mathrm{Cvr}^{m_H}(H)$ and $\mathrm{Act}(H) = \mathrm{Act}^{m_H}(H)$. We now describe the structure of H^l, the lth segment of H. We can write H^l as follows:

$$\begin{aligned} H^l = {} & S(p_1, l) \circ C(p_1, l; H) \circ S(p_2, l) \circ C(p_2, l; H) \circ \cdots \circ S(p_k, l) \circ C(p_k, l; H), \\ & \text{where } \{p_1, p_2, \ldots, p_k\} = \mathrm{Act}^l(H). \end{aligned} \tag{8}$$

$S(p, l)$, the lth solo segment of p, was already defined in Section 3. 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 (1) 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:

$$AW_v^l = \{p \in \mathrm{Act}^l(H)\colon ce(p, l) \text{ is a write to } v\}. \tag{9}$$

We now describe the structure of $C(p, l)$ in detail. For each $m \leq l$, $ce(p, m)$ 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 6(a).) Thus, as described before, we deploy a process q from CW_v^m, and let q execute its invocation event $ie(q, m)$ 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 6(b) and (c) are contained in covering segments, which have been omitted from the figure for simplicity. For example, $ie(q, m)$ and $ie(q', l)$ 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:

$$\begin{aligned} RW_v^m &= \{q \in CW_v^m\colon q \text{ is not deployed in } H\} \\ &= \{q \in CW_v^m\colon cp(q) = \bot\}. \end{aligned} \tag{10}$$

When we consider multiple regular computations, we also write $CW_v^m(H)$, $AW_v^m(H)$, $RW_v^m(H)$, $ci(p; H)$, $cv(p; H)$, and $cp(p; H)$ in order to specify the relevant computation H. Note that, because our proof involves erasing processes from existing computations, each induction step may alter previously constructed segments. For example, if we refer to the computation obtained at the mth induction step as H_m, then in general, the sets CW_v^m, AW_v^m, RW_v^m may vary between H_m and H_{m+1}, and among any further induction step. Similar remark applies to $ci(p)$, $cv(p)$, and $cp(p)$.

The structure of a regular computation, explained so far, is depicted in Figure 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 one-shot mutual exclusion system. A computation H in C is *regular* if and only if it satisfies the following.

H can be written as $H = H^1 \circ H^2 \circ \cdots \circ H^{m_H}$, where H^m is called the mth *segment* of H. Segment H^m consists of the events appended at the mth 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 (2)–(4), (6), (7), (9), and (10). Segment H^m can be written as in (8). 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 $j \le m$ 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 $j \le m$, 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 ($1 \le m \le m_H$) 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 (8), $p \in \mathrm{Act}^m(H)$ 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 $m \le j \le m_H$, 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 Property A. 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 9.

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.

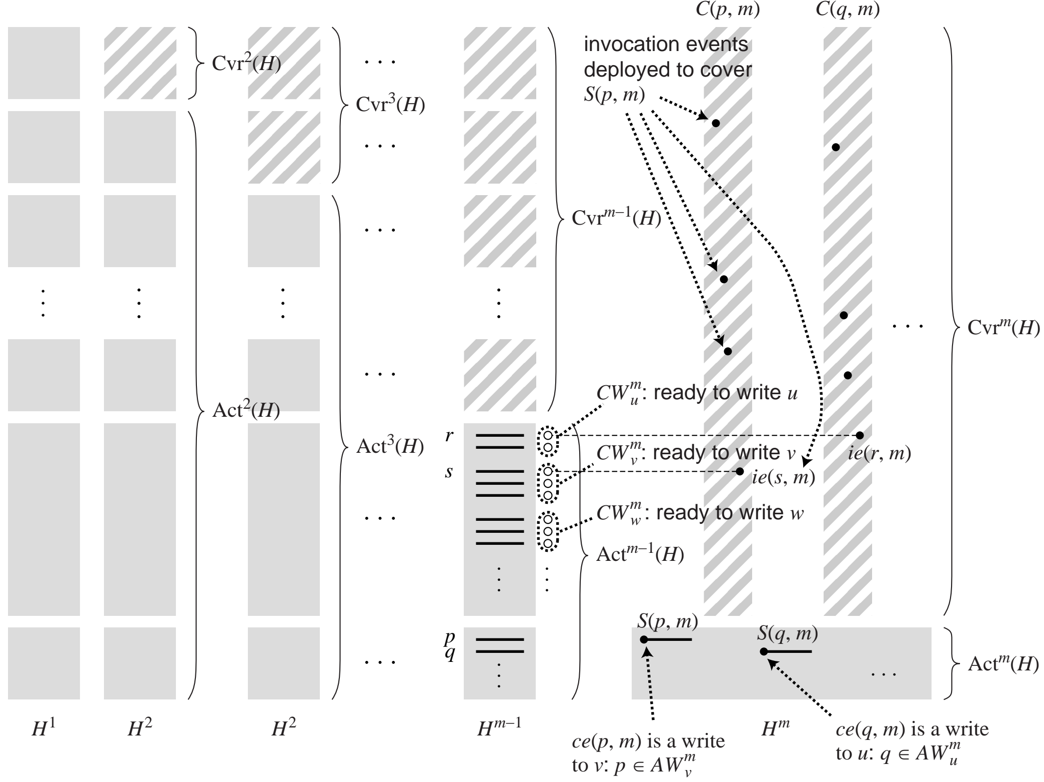


Figure 8: The structure of a regular computation. The computation is depicted as a collection of boxes, where the horizontal axis represents sequential order and the vertical axis represents different processes. A grey-filled box represents a collection of solo segments, and a striped box represents a collection of covering segments. 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. (Note that $\mathrm{Cvr}^1(H)$ is shown to be empty, since no processes are selected for covering in the first induction step — each process p merely executes $Enter_p$.) We assume that the covering strategy is used at the mth 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 mth 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 mth 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 $S(p,m)$.

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. Condition (iii) clearly prohibits some process q from reading v *for the first time* within segment m or later. Now, 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 (8)), 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.

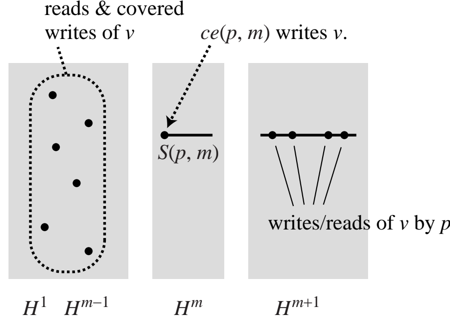


Figure 9: The single-writer case. We assume that $p \in \mathrm{Act}^m(H)$, $ce(p,m)$ is a write to v (*i.e.*, $p \in AW_v^m$), and that CW_v^m is empty. Segments H^1, H^2, $\ldots$, H^{m-1} may contain reads and covered writes of v. In segment H^m and later segments, p is the only process that may access v.

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. □

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 (9), 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.[10] 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 the existence of some positive integer value $c = c(N)$, satisfying

$$c = \Theta(\log N), \tag{11}$$

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 exclusion system, and hence we consider c as a fixed constant.)

Unfortunately, this bound on $|CW_v^m|$ is still insufficient, since processes in CW_v^m may also be erased in induction steps beyond the mth. (See Figure 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, on the average, each induction step erases at most c processes from CW_v^m. Thus, at the mth step, we select $c \cdot (c - m)$ *additional* processes for covering, since we have at most $c - m$ induction steps beyond the mth.

It follows that we need to select at least $c \cdot (|AW_v^m| + c - m)$ covering processes for CW_v^m, at the mth induction step. Since H has taken a total of m_H induction steps to construct, $c \cdot (m_H - m)$ processes may have already been erased from CW_v^m (c processes for each step between $(m+1)^{\mathrm{st}}$ and ${m_H}^{\mathrm{th}}$). Therefore, we define $\mathrm{req}(m, v; H)$, the *required number* of covering processes for $AW_v^m(H)$, as $c \cdot (|AW_v^m(H)| + c - m) - c \cdot (m_H - m)$:

$$\mathrm{req}(m, v; H) = c \cdot (|AW_v^m(H)| + c - m_H). \tag{12}$$

[10] 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.

To capture the notion of how "deficient" a computation is of covering processes, we use the notion of a "rank," defined next. Consider a fixed 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:

$$\pi(m, v; H) = \begin{cases} \max\{0,\ \mathrm{req}(m, v; H) - |CW_v^m(H)|\}, & \text{if } AW_v^m(H) \ne \{\} \ \wedge\ CW_v^m(H) \ne \{\}; \\ 0, & \text{otherwise.} \end{cases} \tag{13}$$

When there is no possibility of confusion, we use $\mathrm{req}(m, v)$ and $\pi(m, v)$ as shorthands for $\mathrm{req}(m, v; H)$ and $\pi(m, v; H)$. 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:

$$\pi_{\max}(H) = \max_{1 \le m \le m_H,\ v \in V} \pi(m, v; H); \tag{14}$$

$$\begin{aligned} \pi(H) &= \sum_{1 \le m \le m_H,\ v \in V} \pi(m, v; H) \\ &= \sum_{(m,v):\ \text{covering pair in } H} \pi(m, v; H). \end{aligned} \tag{15}$$

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 (*i.e.*, $\pi(m, v; H)$ is zero for all m and v). Within a single 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 $\pi(m, v)$.

The following lemma ensures that a regular computation with a "low" maximum rank has "enough" reserve writers, for each covering pair.

Lemma 1 *Consider a regular computation H in C with induction number m_H. Assume the following:*

- $m_H \le c - 2$, *and* (16)
- $\pi_{\max}(H) \le c$. (17)

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 (17), and by the definitions of $\pi_{\max}$ and 'req' (given in (12)–(14)), we have $\mathrm{req}(j, w; H) - |CW_w^j| \le c$, and hence,

$$|CW_w^j| \ge \mathrm{req}(j, w; H) - c = c \cdot (|AW_w^j| + c - m_H - 1) > c \cdot |AW_w^j|, \tag{18}$$

where the last inequality follows from (16).

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 $k \ge j$ and $q \in AW_w^j \cap \mathrm{Act}^k(H)$ (see Figure 6). Thus,

$$\begin{aligned} |RW_w^j| &= |CW_w^j| - \sum_{k=j}^{m_H} |AW_w^j \cap \mathrm{Act}^k(H)| \\ &\ge |CW_w^j| - (m_H - j + 1) \cdot |AW_w^j| \\ &\ge |CW_w^j| - m_H \cdot |AW_w^j| \\ &> (c - m_H) \cdot |AW_w^j| && \{\text{by (18)}\} \\ &> |AW_w^j|, && \{\text{by (16)}\} \end{aligned}$$

/* First, construct a regular computation H 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 $\langle Enter_p \rangle$.
Each process is indexed as a number between 1 and N, *i.e.*, we have $P = \{1, 2, \ldots, N\}$. */

1: Construct $H_1 := S(1,1) \circ S(2,1) \circ \cdots \circ S(N,1)$ $(= \langle Enter_1, Enter_2, \ldots, Enter_N \rangle)$;
2: $m := 1$;
3: **while** $(m \leq c - 2) \wedge (|\mathrm{Act}(H_m)| \geq 2)$ **do**
/* Loop invariant: H_m is a regular computation with induction number m and maximum rank zero (*i.e.*, $\pi_{\max}(H) = 0$). */
4: Construct $H_{m+1} :=$ ApplySingleInductionStep(H_m);
5: $m := m + 1$
od

function ApplySingleInductionStep(H: **regular computation**): **regular computation**
6: $m :=$ (induction number of H);
7: $n := |\mathrm{Act}(H)|$;
/* We may assume $m \leq c - 2$, $n \geq 2$, and $\pi_{\max}(H) = 0$. */
8: $(F, Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}}) :=$ EliminateConflict(H);
/* F is a regular computation with induction number m, satisfying (19) below. */

$$\begin{cases} \pi_{\max}(F) \leq c \\ \mathrm{Act}(F) = Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}} \quad \text{(disjoint union)} \\ |Z^{\mathrm{Act}}| \geq \dfrac{(c-2)(n-1)}{48c^2(c-1)(2m+1)} \end{cases} \tag{19}$$

9: Construct $E :=$ BuildNextSegment$(F, Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$; /* Build the next $(m+1)^{\mathrm{st}}$ segment E. */
10: $G := F \circ E$;
/* G is a regular computation with induction number $m + 1$, such that $\pi_{\max}(G) = 0$ and $\mathrm{Act}(G) = Z^{\mathrm{Act}}$. */
11: **return** G
end

Figure 10: High-level description of the lower bound construction: the main algorithm.

which completes the proof. □

4.3 Detailed Proof Overview

Due to its complexity, our lower-bound proof is sketched in Figures 10, 11, 15 and 16 as a pseudo-algorithm. We now give a detailed account of each step involved.

Lines 1–5 comprise the "main" part of our pseudo-algorithm. Initially (lines 1 and 2), 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 $\langle Enter_p \rangle$.

We then repeatedly apply an induction step (lines 4 and 5) until the loop condition (line 3) becomes false. Lines 4 and 5 comprise the $(m+1)^{\mathrm{st}}$ induction step. Here, 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 applying a single induction step, we erase some processes in H_m and append a new $(m+1)^{\mathrm{st}}$ segment. As a result, we obtain a regular computation H_{m+1} with a maximum rank of zero and induction number $m + 1$ (*i.e.*, each process in $\mathrm{Act}(H_{m+1})$ executes $m + 1$ solo segments in H_{m+1}). Moreover, as explained shortly (see (20) below), $\mathrm{Act}(H_{m+1})$ consists of $\Omega(n/c^3)$ $(= \Omega(n/\log^3 N))$ processes.[11]

By repeating the induction step, we construct a series of regular computations $H_1, H_2, \ldots, H_m$. The induction terminates when either $m = c - 1$ is established or only one active process is left.[12] In the

[11] We use $\log n$ to denote $\log_2 n$ (base-2 logarithm), and use $\log^k n$ to denote $(\log_2 n)^k$.

former case, by (11), we have $m = \Theta(\log N)$. In the latter case, by combining the inequality $|\mathrm{Act}(H_{k+1})| = \Omega(|\mathrm{Act}(H_k)|/\log^3 N)$ over $k = 1, 2, \ldots, m-1$, and using $|\mathrm{Act}(H_m)| = 1$, we can show $m = \Omega(\log N/\log\log N)$. Therefore, in either case, we have established $m = \Omega(\log N/\log\log N)$. Since each active process in H_m executes m solo segments (and hence, m critical events) in H_m, we have our lower bound.

Each induction step, shown as function ApplySingleInductionStep, starts with a regular computation H (with induction number m) and yields another regular computation G with induction number $m+1$. This function in turn calls the following two functions.

- Function EliminateConflict determines which process may generate *conflicts* if its next ($(m+1)^{\mathrm{st}}$) critical event is appended to H, and then eliminates possible conflicts by erasing some of these processes. (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 G.) As a result, we obtain a computation F with induction number m, in which all "troublesome" processes are already erased. In addition, EliminateConflict also determines whether each surviving active process in F should remain active or become a covering process. The former set of processes are returned as Z^{Act}, the latter as Z^{Cvr}.
- Function BuildNextSegment actually builds the next segment E. Since $\mathrm{Act}(G) = Z^{\mathrm{Act}}$, by (19) (given at line 8), we have $|\mathrm{Act}(G)| \geq \big[(c-2)(n-1)\big]/\big[48c^2(c-1)(2m+1)\big]$. Combined with $m \leq c-2$ and $n \geq 2$ (and hence $n-1 = \Theta(n)$), this in turn implies

$$|\mathrm{Act}(G)| = \Omega(n/c^3), \tag{20}$$

as claimed before.

We now describe function EliminateConflict (Figure 11), which constitutes the centerpiece of our proof and is formally described in Lemma 8 in Appendix B. We are given a regular computation H with induction number m, in which n active processes participate. Since EliminateConflict is called only from line 8, we may also assume $m \leq c-2$, $n \geq 2$, and $\pi_{\max}(H) = 0$. Our goal is to erase some (active and covering) processes, partition the remaining active processes into Z^{Act} and Z^{Cvr}, and yield a regular computation F which satisfies (19). Towards this goal, EliminateConflict executes the following four steps.

Step 1: choosing readers or writers. Every process in $\mathrm{Act}(H)$ has executed its first m solo segments, and hence is ready to execute its $(m+1)^{\mathrm{st}}$. For each $p \in \mathrm{Act}(H)$, we define its "next" critical event, denoted e_p, to be $ce(p, m+1)$, the event p is ready to execute after H.

By the Exclusion property, it follows that at most one process p in $\mathrm{Act}(H)$ may execute CS_p after H. Consider the remaining $n-1$ active processes (or all n active processes, if there exists no such process). They 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 (line 14), and erase all other active processes (line 15). Clearly, we have $|Y| \geq (n-1)/2$ (see Figure 12). Although erasing up to half of the active processes is not strictly necessary, this greatly reduces the amount of bookkeeping required in later steps.

When we erase an active process p, we erase all of its solo segments $S(p, j)$ (for $1 \leq j \leq m$), as well as its covering segments $C(p, j)$. (Recall that p's covering segment consists of invocation events by *other* processes, which are deployed to cover writes by p. See (8) and Figure 8.) Thus, processes that are deployed to cover p's solo segments are turned into reserve processes. Also note that, by definition, erasing an active process may never increase the maximum rank, and hence we have $\pi_{\max}(H') = 0$. (This is formally proved in Lemma 5 in Appendix B; the formal definition of the "erasing operator" $erase_p$, used in Lemma 5, is given right after Lemma 2.)

[12] In Appendix B, we actually terminate the induction if fewer than seven active processes are left (condition (133) of Lemma 8). This allows us to eliminate certain boundary cases; in particular, it is easier to show that we do not accidently erase *all* active processes. (See footnote 17.) Clearly, this does not affect our asymptotic lower bound.

function EliminateConflict(H: **regular computation**):
  (**regular computation**, **set of processes**, **set of processes**)   /* Lemma 8 */

/* This function eliminates conflicts by erasing some processes.
We examine the next solo segment $S(p, m+1)$ for each $p \in \text{Act}(H)$, in order to identify any conflict (information flow) that may arise should we append $S(p, m+1)$ to H.
We then erase some active and covering processes in order to eliminate all such conflicts. The resulting computation F also has induction number m (*i.e.*, the next solo segments are not yet appended); $\text{Act}(F)$ is partitioned into two disjoint sets, Z^{Act} and Z^{Cvr}. Z^{Act} (Z^{Cvr}) will be the set of active (covering) processes in G, constructed in line 10. */

12: $m :=$ (induction number of H);
13: $n := |\text{Act}(H)|$;
  /* We may assume $m \le c-2$, $n \ge 2$, and $\pi_{\max}(H) = 0$. */

  /* **Step 1.** Among the active processes, choose the larger of either the "readers" or the "writers," and erase the rest — see Figure 12. */

14: For each $p \in \text{Act}(H)$, let e_p be its next critical event, $ce(p, m+1)$;
  By the Exclusion Property, there exists at most one process p such that $e_p = CS_p$;
  Choose a set $Y \subseteq \text{Act}(H)$ satifying the following:

$$\begin{cases} e_p \neq CS_p, \quad \text{for all } p \in Y \\ |Y| \ge (n-1)/2 \\ \big(\forall p : p \in Y :: e_p \text{ is a read event}\big) \quad \vee \quad \big(\forall p : p \in Y :: e_p \text{ is an atomic write event}\big); \end{cases}$$

15: Erase all processes in $\text{Act}(H) - Y$, and denote the resulting computation as H';
  /* H' is a regular computation such that $\text{Act}(H') = Y$ and $\pi_{\max}(H') = 0$. */

  /* **Step 2 (Chain Erasing).** Eliminate conflicts *between* Y and $\text{Cvr}(H')$ ($= \text{Cvr}(H)$), while maintaining the invariant (maximum rank $< c$). */

16: Construct a conflict map K: $Y \to P(H') \cup \{\bot\}$, as follows:
    For each $p \in Y$, let $v_{\text{ce}}(p)$ be the variable accessed by p's next critical event e_p;
    If $v_{\text{ce}}(p)$ has a single writer (see Figure 9) q in H', and if $q \neq p$, then let $K(p) = q$; otherwise, let $K(p) = \bot$;

17: Define $CE := \{K(p)\colon p \in Y \ \wedge\ K(p) \in \text{Cvr}(H')\}$;
18: Construct $H'' := \text{ChainErase}(H', CE)$;   /* Apply chain erasing (Lemma 7). */
19: Define $Y' := \text{Act}(H'')$;
  /* H'' is a regular computation with induction number m, such that $\pi_{\max}(H'') < c$, $|\text{Act}(H'')| \ge |Y| - |CE|/(c-1)$, and $P(H'') \cap CE = \{\}$. */

  /* **Step 3.** Eliminate conflicts *between* the next critical events (by processes in Y'), and pre-existing *write* events in H'' by processes in Y'. */

20: Construct an undirected graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y', as follows: for each process p in Y', we introduce edge $\{p, K(p)\}$ if $K(p) \in Y'$ holds;
21: Apply Turán's Theorm (Theorem 2) and obtain an independent set $Z \subseteq Y'$;
  /* We have $|Z| \ge |Y'|/3$ and $p \neq K(q)$, for all p and q in Z. */

Figure 11: High-level description of the lower bound construction: eliminating conflicts. (Continued on the next page.)

Construction of the conflict map K. The next critical events by processes in Y may generate conflicts, either with pre-existing events in H' or among themselves. We can partition these conflicts into four categories: **(A)** a conflict *between* some next critical event (by a process in Y) and a pre-existing event by some covering process (*i.e.*, one in $\text{Cvr}(H')$), **(B)** a conflict *between* some next critical event (by a process in Y) and a pre-existing *write* event by some process in Y, **(C)** a conflict *between* some next critical *write* event (by a process in Y) and some pre-existing *read* event by a process in Y,[13] and **(D)** conflicts *among* the next critical events. Steps 2 and 3 eliminate conflicts of types A and B, respectively; Step 4 eliminates both types C and D.

[13] At first sight, it may seem unclear why a write following a read should be considered a conflict. However, this violates Condition R4(ii), as discussed earlier.

/* **Step 4.** Eliminate all remaining conflicts. */

22: Group processes in Z depending on the variables accessed by their next critical events;
For each $v \in V$, define Z_v to be the set of processes in Z that access v in their next critical events:

$$Z_v = \{p \in Z \colon v_{\mathrm{ce}}(p) = v\};$$

23: 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

$$\begin{array}{rcl} V_{\mathrm{HC}} & = & \{v \in V \colon |Z_v| \geq 4c^2\}, \quad \text{and} \\ V_{\mathrm{LC}} & = & \{v \in V \colon 0 < |Z_v| < 4c^2\}; \end{array}$$

Similarly, define P_{HC} and P_{LC}, the set of processes that experience "high contention" and "low contention," respectively, as

$$\begin{array}{rcccl} P_{\mathrm{HC}} & = & \bigcup_{v \in V_{\mathrm{HC}}} Z_v & = & \{p \in Z \colon e_p \text{ accesses some variable in } V_{\mathrm{HC}}\}, \quad \text{and} \\ P_{\mathrm{LC}} & = & \bigcup_{v \in V_{\mathrm{LC}}} Z_v & = & \{p \in Z \colon e_p \text{ accesses some variable in } V_{\mathrm{LC}}\}; \end{array}$$

24: **if** (all next critical events by Z are reads) **then**
/* Note that, by Step 1 above, either all next critical events by Z are reads, or they are all writes. */

/* **Case 1:** readers only. */
25: Define $Z^{\mathrm{Act}} := Z$ and $Z^{\mathrm{Cvr}} := \{\}$

26: **elseif** ($|P_{\mathrm{HC}}| < |Z|/2$) **then**

/* **Case 2:** erasing strategy. */
27: Construct a subset X of P_{LC} that contains exactly one process from each Z_v (for each $v \in V_{\mathrm{LC}}$);
28: Construct an undirected graph $\mathcal{H} = (X, E_{\mathcal{H}})$, where each vertex is a process in X, as follows: for each pair $\{p, q\}$ of different processes in X, we introduce edge $\{p, q\}$ if p reads $v_{\mathrm{ce}}(q)$ in H'' (see Figure 14);
/* The average degree of X is at most $2m$. */
29: Apply Turán's Theorem (Theorem 2) and obtain an independent set X' such that $|X'| \geq |X|/(2m+1)$;
30: Define $Z^{\mathrm{Act}} := X'$ and $Z^{\mathrm{Cvr}} := \{\}$

else

/* **Case 3:** covering strategy; */
31: For each $v \in V_{\mathrm{HC}}$, choose exactly $\lfloor |Z_v|/c^2 \rfloor - 1$ processes and insert them into Z^{Act}; insert the rest into Z^{Cvr}

fi

32: Erase all processes in $Y' - (Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}})$ from H'', and denote the resulting computation as F;
/* F is a regular computation with induction number m, satisfying (19) (see line 8). */

33: **return** $(F, Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$
end

Figure 11: High-level description of the lower bound construction: eliminating conflicts, continued.

We begin by considering conflicts of types A and B together. In order to determine "who conflicts with whom," we define a "conflict mapping" K, defined over Y (line 16). For each $p \in Y$, we define $K(p) = q$ if p's next critical event e_p accesses variable v, and if q ($\neq p$) is the single writer of v in H (as depicted in Figure 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.

Step 2: the chain erasing. In order to eliminate type-A conflicts, we define CE, the "Covering processes to be Erased," as $\{K(p) \colon p \in Y \text{ and } K(p) \in \mathrm{Cvr}(H')\}$ (line 17). We then apply the chain erasing procedure (line 18), described in detail later, in order to erase processes in CE. Erasing covering processes (in CE) is not as simple as erasing active processes, for the following two reasons.

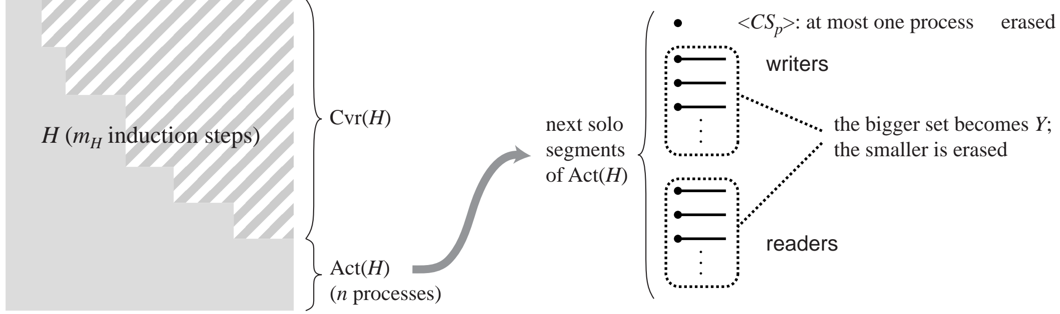


Figure 12: Construction of Y from Act(H). As in Figure 8, the grey-filled region represents a collection of solo segment(s), and the striped region represents a collection of covering segment(s). At most one process p in Act(H) may enter its critical section (*i.e.*, execute CS_p) after H. We partition the rest of Act(H) into readers and writers, let Y be the larger of the two, and erase other active processes.

First, if a process $q \in CE \cap CW_v^j$ (for some j and v) is already deployed to cover another process, then we have to find some reserve process $r \in RW_v^j$, and "exchange" the role of q and r, before erasing q (see Figure 7). We also have to show that RW_v^j is nonempty, so that we can choose r.

Second, we have to ensure that we do not erase "too many" covering processes, because otherwise we may not have enough reserve processes in later induction steps. We solve this problem by ensuring that the maximum rank remains lower than c. Towards this goal, we may have to erase some active processes, even though they do not cause conflicts. (Recall that a large rank indicates that we may not have enough covering processes, *i.e.*, we may have too *many* active processes.)

Later in this section, we give a detailed account of the chain erasing procedure, depicted in function ChainErase (Figure 16). This function returns a computation H'' with maximum rank at most $c-1$, in which all processes in CE are erased (along with at most $|CE|/(c-1)$ active processes). We define Y' to be the set of surviving active processes (line 19). Thus, we have

$$|Y'| \geq |Y| - |CE|/(c-1). \tag{21}$$

By definition (given in (12)), for each covering pair (j, v), req(j, v) is reduced by c when we later append the new $(m+1)^{\text{st}}$ segment (E) at line 10. Hence, the rank $\pi(j, v)$, being less than c in H'', is reduced to zero after appending E. It follows that all covering pairs in H'' will have zero rank in the extended computation G (line 11).

Step 3: eliminating conflicts of type B. We now consider Step 3 (lines 20 and 21). In order to eliminate conflicts of type B, we construct 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. Clearly, we introduce at most $|Y'|$ edges in total. The construction of $\mathcal{G}$ is shown in Figure 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 of conflicts of type B. Toward this goal, we use Turán's Theorem [33], stated below.

Theorem 2 (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.* □

Since $\mathcal{G}$ has $|Y'|$ vertices and at most $|Y'|$ edges, by applying Turán's Theorem, we can obtain an independent set $Z \subseteq Y'$ with at least $|Y'|/3$ processes. (All processes in $Y' - Z$ are eventually erased in line 32.)

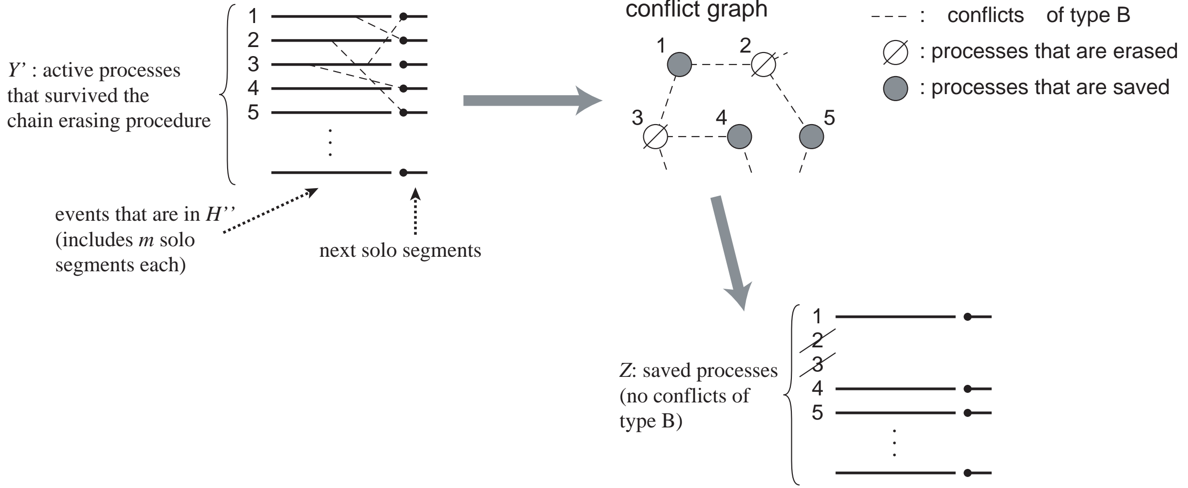


Figure 13: Construction of the "conflict graph" $\mathcal{G}$. For simplicity, covering processes are not shown in this figure. In this figure, we assume $K(1) = 3$, $K(2) = 1$, $K(3) \notin Y'$, etc.

Step 4: eliminating conflicts of types C and D. The processes in Z collectively execute $|Z|$ next critical events. We may partition Z into Z_v for each variable v, depending on the variables that are accessed by these critical events (line 22). Among these variables, we also 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 into P_{HC} and P_{LC}, depending on whether their next critical events access a variable in V_{HC} or V_{LC} (line 23).

Because Z consists of active processes, we can erase any process in Z and preserve regularity. We now have to eliminate conflicts of types C and D (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 $(m+1)^{\text{st}}$ segment. As explained before, these selected processes comprise subsets Z^{Act} and Z^{Cvr} of Z, respectively. Processes in $Z - (Z^{\text{Act}} \cup Z^{\text{Cvr}})$ (if any) are simply erased (see line 32).

Recall that, thanks to Step 1, Z consists of either all "writers" or all "readers." We consider three cases.

Readers only. Consider a variable v that is read by the next critical event of some process p in Z. Note that H 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 q has been erased in Step 2 (a type-A conflict). If, on the other hand, $q \in \text{Act}(H)$, and if q survives Steps 1 and 2, then we have $q \in Y'$. Hence, $\{p, q\} = \{p, K(p)\}$ is an edge in $\mathcal{G}$, and hence $p \in Z$ implies $q \notin Z$ (a type-B conflict; erased in Step 3).

Therefore, we can simply define $Z^{\text{Act}} = Z$ and $Z^{\text{Cvr}} = \{\}$ (line 25). Later, in the extended computation G, the next critical event by each 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 retain at least half of the processes in Z. Since every variable in V_{LC} is accessed by at most $4c^2$ different next critical events, V_{LC} contains at least $|P_{\text{LC}}|/4c^2$ variables. By selecting one process for each such variable, we can create a set X of active processes, such that $|X| \geq |P_{\text{LC}}|/4c^2$, in which each next critical event accesses a distinct variable (line 27).

We want each process $p \in X$ to become the single writer of $v_{\text{ce}}(p)$. Note that H'' does not contain a single writer of $v_{\text{ce}}(p)$, as discussed above in the "readers only" case. In order to eliminate type-C conflicts,

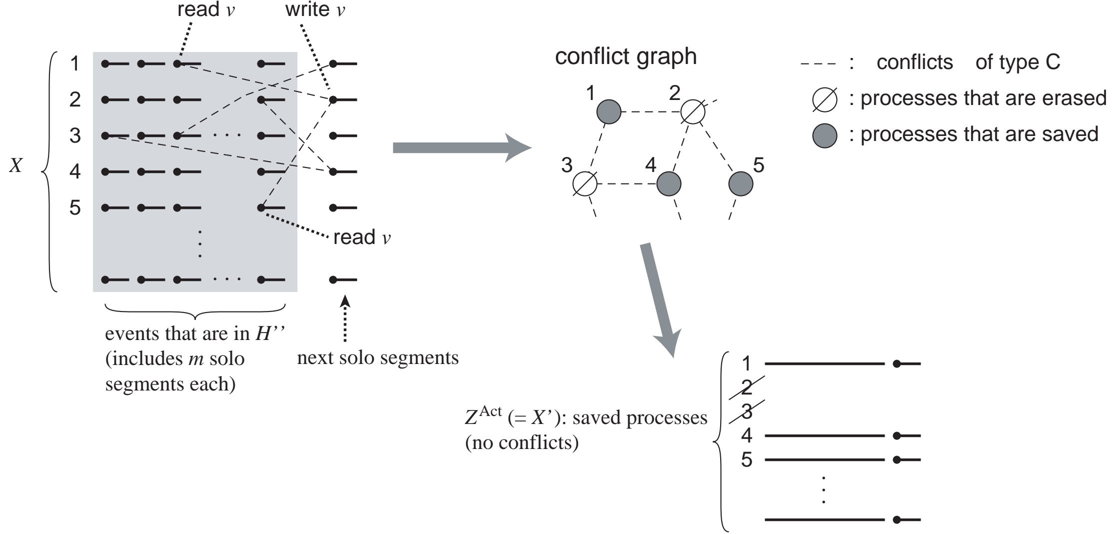


Figure 14: Erasing strategy. For simplicity, covering processes are not shown in this figure. In this figure, we assume that the next critical event of process 2 writes v (*i.e.*, $v_{\text{ce}}(2) = v$), and that processes 1 and 5 read v in H''.

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 14.

Since each process p in X executes m critical events in H, p may read at most m different variables. For each variable v read by p, we introduce edge $\{p, q\}$ if $v = v_{\text{ce}}(q)$ (line 28). Since each $v_{\text{ce}}(q)$ is distinct (by the construction of X), we introduce at most m edges per each process in X. By applying Theorem 2 again, we can construct a subset X' of X without any conflicts, such that $|X'| \geq |X|/(2m+1)$ (line 29). We then define $Z^{\text{Act}} = X'$ and $Z^{\text{Cvr}} = \{\}$ (line 30). Every process $p \in Z^{\text{Act}}$ will become the single writer of $v_{\text{ce}}(p)$ in the extended computation G. Note that, since $|P_{\text{LC}}| \geq |Z|/2$, we have the following:

$$|Z^{\text{Act}}| = |X'| \geq \frac{|X'|}{2m+1} \geq \frac{|P_{\text{LC}}|}{4c^2(2m+1)} \geq \frac{|Z|}{8c^2(2m+1)}.$$

Covering strategy. Assume that Z consists only of "writers," and that P_{HC} is larger than P_{LC}. In this case, we first erase P_{LC}. Every next critical event by a process in P_{HC} writes a variable in V_{HC}. We now apply the covering strategy (see Figure 6) to each variable in V_{HC}. Since each variable $v \in V_{\text{HC}}$ is written by at least $4c^2$ processes in P_{HC}, we can choose active writers (*i.e.*, processes in Z^{Act}) and covering writers (*i.e.*, processes in Z^{Cvr}), satisfying the following: **(i)** the number of all active writers is $\Omega(|Z|/c^2)$ (specifically, at least $|P_{\text{HC}}|/2c^2$), and **(ii)** for each variable $v \in V_{\text{HC}}$, the rank $\pi(m+1, v; G)$ is zero, where G is the extended computation to be constructed,

Formally, for each $v \in V_{\text{HC}}$, assume that $k(v)$ $(\geq 4c^2)$ processes in P_{HC} write v. Among these processes, we choose $a(v)$ processes to become active writers, and the rest to become covering writers, where

$$a(v) = \lfloor k(v)/c^2 \rfloor - 1. \tag{22}$$

As a result, we have the following inequalities, from which Conditions (i) and (ii) follow.

$$\begin{aligned} a(v) &> \frac{k(v)}{c^2} - 2 && \{\text{by (22)}\} \\ &\geq \frac{k(v)}{c^2} - \frac{k(v)}{2c^2} = \frac{k(v)}{2c^2} && \{\text{since } k(v) \geq 4c^2\} \end{aligned}$$

$$
\begin{aligned}
|CW_v^{m+1}| &= k(v) - a(v) \\
&\geq c^2 \cdot (a(v)+1) - a(v) && \{\text{by } (22)\} \\
&= (c^2 - 1) \cdot a(v) + c^2 \\
&> c \cdot a(v) + c^2 && \{c^2 - 1 > c \text{ for } c \geq 2\} \\
&> c \cdot (a(v) + c - m_G) && \{m_G = m+1\} \\
&= \text{req}(m+1, v; G) && \{\text{by the definition of "req," given in } (12)\}
\end{aligned}
$$

The collection of all active writers becomes Z^{Act}, and the collection of all covering writers becomes Z^{Cvr}. (Hence, we have $Z^{\text{Act}} \cup Z^{\text{Cvr}} = P_{\text{HC}}$.) Note that, since all next writes are properly covered, type-C conflicts do not arise in this case.

Analysis of EliminateConflict**.** As the last step of EliminateConflict, we erase all active processes not in $Z^{\text{Act}} \cup Z^{\text{Cvr}}$ (line 32), and return the resulting computation (line 33).

We now claim that F satisfies condition (19), given at line 8 of Figure 10. Since erasing active processes cannot increase any rank, we have $\pi_{\max}(F) \leq \pi_{\max}(H'') < c$. Clearly, we also have $\text{Act}(F) = Z^{\text{Act}} \cup Z^{\text{Cvr}}$. Finally, note that we have one of the following:

$$
\begin{cases}
|Z^{\text{Act}}| = |Z| & \text{(Readers only)}, \\
|Z^{\text{Act}}| \geq \dfrac{|Z|}{8c^2(2m+1)} & \text{(Erasing strategy)}, \\
|Z^{\text{Act}}| \geq \dfrac{|Z|}{4c^2} & \text{(Covering strategy)}.
\end{cases}
$$

Combining this with $|Y| \geq (n-1)/2$ (see line 14), $|CE| \leq |Y|$ (by definition; see line 17), $|Y'| \geq |Y| - |CE|/(c-1)$ (by (21)), and $|Z| \geq |Y'|/3$ (see line 21), the third line of (19) follows. (For a detailed analysis, we refer the reader to Claim 2 in the proof of Lemma 8.)

Construction of the next segment. We now describe function BuildNextSegment, which is depicted in Figure 15 and formally described in Lemma 9 in Appendix B.

During the execution of BuildNextSegment, variable E holds the partially constructed prefix of the next segment. As defined in (8) and illustrated in Figure 8, the next segment can be constructed by alternating next solo segments $S(p, m+1)$ and next covering segments $C(p, m+1)$, for each $p \in Z^{\text{Act}}$. Each $C(p, m+1)$ is in turn constructed by deploying appropriate covering processes so that any write (critical or noncritical) contained in $S(p, m+1)$ is properly covered (see Figure 6).

To facilitate this, BuildNextSegment uses a boolean array *deployed*, which indicates whether each deployable process is deployed so far. The value of *deployed*$[p]$ is meaningful only if p is in either $\text{Cvr}(H)$ or Z^{Cvr} — in the latter case, p eventually becomes a covering process in the extended computation $G = H \circ E$.

BuildNextSegment starts with lines 34–38, which initialize variables to be used in the following loop. For each $p \in Z^{\text{Act}}$, appending its next solo segment is straightforward (line 40). In order to construct its next covering segment $C(p, m+1)$, we first iterate over each j $(1 \leq j \leq m)$ (lines 41–45) and check if the covering strategy was used at the jth induction step (line 42).

In particular, if the condition stated at line 42 is true, then p's j^{th} critical event writes to v, and hence $S(p, m+1)$ may (noncritically) write to v. In order to cover this, we choose a reserve process q from $RW_v^j(H)$ and deploy it (lines 43–45). (On the other hand, if $CW_v^j(H)$ is empty, then p is the single writer of v, and hence no covering is necessary.)

We now claim that we can always choose an undeployed process q at line 43. Consider a fixed covering pair (k, u). Because BuildNextSegment is called only from line 9, we may assume $m \leq c-2$ and $\pi_{\max}(H) \leq c$. Hence, by Lemma 1, we have $|RW_u^k(H)| > |AW_u^k(H)|$. Also, lines 43–45 can be executed with $(j, v) = (k, u)$ at most $|AW_u^k(H)|$ times (to be exact, once for each $p \in AW_u^k(H) \cap Z^{\text{Act}}$). Since every process in $RW_u^k(H)$ is initially undeployed by definition, the claim follows.

```
function BuildNextSegment(H: regular computation, Z^Act: set of processes, Z^Cvr: set of processes):
    computation                                                          /* Lemma 9 */
variables deployed: array[1..N] of boolean
34: m := (induction number of H);
    /* We may assume m ≤ c − 2 and π_max(H) ≤ c. */
35: For each p ∈ Act(H), let e_p be its next critical event, ce(p, m + 1);
36: E := ⟨⟩;
37: for each p ∈ Cvr(H) ∪ Z^Cvr do
38:     deployed[p] := if (p is deployed in H) then true else false
    od;
39: for each p ∈ Z^Act do
40:     E := E ∘ S(p, m + 1);       /* Append p's next solo segment. */
41:     for j := 1 to m do
42:         if (∃v : v ∈ V :: p ∈ AW_v^j(H) ∧ CW_v^j(H) ≠ {}) then
                /* p's j^th critical event writes v; we deploy a reserve process from RW_v^j(H) in order to cover p's
                   (possible) write to v in S(p, m + 1). */
43:             Choose a process q such that q ∈ RW_v^j(H) and deployed[q] = false;
44:             E := E ∘ ⟨ie(q, j)⟩;
45:             deployed[q] := true
        fi od;
46:     if (Z^Cvr ≠ {}) then
            /* Covering strategy was selected in EliminateConflict. */
47:         Let v be the variable written by p;
            Choose a process q such that q ∈ Z^Cvr, ce(q, m + 1) writes v, and deployed[q] = false;
48:         E := E ∘ ⟨ie(q, m + 1)⟩;
49:         deployed[q] := true
        fi
    od;
50: return E
end
```

Figure 15: High-level description of the lower bound construction: building the next segment.

Additionally, if the covering strategy (line 31) was chosen at Step 4 of EliminateConflict, then we also have to cover p's next critical write (e_p) by deploying a process from Z^{Cvr} (lines 46–49). Recall that, if $k(v)$ ($\geq 4c^2$) processes in $Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}$ write v in their next critical events, then exactly $a(v) = \lfloor k(v)/c^2 \rfloor - 1$ of them belong to Z^{Act} (see (22)). Hence, among these $k(v)$ processes, more processes belong to Z^{Cvr} than to Z^{Act}. Therefore, we can always choose an undeployed process q at line 47.

Finally, we can examine all necessary conditions and verify that $H \circ E$ is indeed a regular computation with induction number $m + 1$ and maximum rank zero. The detailed argument, formally presented in Lemma 9, is rather mechanical and is omitted here.

Detailed analysis of the chain erasing procedure. We now describe function ChainErase, which is depicted in Figure 16 and formally described in Lemma 7. The objective of this function is to erase each process in CE by iteration, while preserving the loop invariant $\pi_{\max}(F) < c$. (Variable F holds the intermediate computation.) An example execution is illustrated in Figure 17. Initially, since ChainErase is called only from line 18, F has induction number $m_F = m_{H'} \leq c - 2$ and maximum rank of zero. For each $p \in CE$, we execute lines 53–65: unless p is already erased as a side effect of earlier iterations, we first erase p (lines 54–58), and then apply the "chain erasing" strategy in order to ensure $\pi_{\max}(F) < c$ (lines 59–65).

We now examine lines 54–65 in detail. Since p is a covering process, $p \in CW_v^j(F)$ holds for some covering pair (j, v) (*i.e.*, p was selected to cover v at the jth induction step). If p is not yet deployed in F (*i.e.*, $p \in RW_v^j(F)$), then we can safely erase p and obtain a regular computation (line 58). On the other hand, if

```
function ChainErase(H': regular computation, CE: set of processes):
    regular computation                                              /* Lemma 7 */

    /* We may assume CE ⊆ Cvr(H'), π_max(H') = 0, and m_{H'} ≤ c − 2. */

51: F := H';
52: for each p ∈ CE do
        /* Loop invariant: π_max(F) < c. */
53:     if p ∈ P(F) then        /* Is p not yet erased? */
54:         j := ci(p);    v := cv(p);        /* p ∈ CW^j_v(F) holds. */
55:         if (p ∉ RW^j_v(F)) then        /* Is p deployed in F? */
56:             Choose a process q ∈ RW^j_v(F);
57:             Exchange p and q from F; let the resulting computation be F
            fi;
58:         Erase p from F; let the resulting computation be F;
59:         while π_max(F) = c do
60:             There exists exactly one covering pair (j, v) satisfying π(j, v; F) = c; choose a process r from AW^j_v(F);
61:             if (r is a deployed covering process in F) then
62:                 k := ci(r);    u := cv(r);        /* r ∈ CW^k_u(F) holds. */
63:                 Choose a process q ∈ RW^k_u(F);
64:                 Exchange r and q from F; let the resulting computation be F
                fi;
65:             Erase r from F; let the resulting computation be F.
    od fi od;
66: return F
end
```

Figure 16: The chain erasing procedure.

p is already deployed in F to cover another process, then we must first find some reserve process $q \in RW^j_v(F)$, and "exchange" the roles of p and q (lines 56 and 57), before we can erase p at line 58. (See Figure 7; this "exchange and erase" strategy is formally described in Lemma 6.)

As shown shortly, $\pi_{\max}(F) < c$ is always true at lines 56–58. Hence, by Lemma 1, we have $|RW^j_v(F)| > |AW^j_v(F)| \geq 0$. It follows that we can always find an (undeployed) reserve process q at line 56.

Although we can erase p and preserve regularity, erasing p reduces $|CW^j_v|$ by one and hence may increase $\pi(j,v)$. If $\pi(j,v)$ is increased to c, then our loop invariant is violated. Informally, a high rank is problematic because there may not be enough reserve processes to continue further induction steps. Note that $\pi(j,v)$ satisfies the following properties, by (12) and (13).

- Erasing a process from CW^j_v increases $\pi(j,v)$ by at most one. (23)
- If we erase a process from AW^j_v while $\pi(j,v) = c$ holds, then $\pi(j,v) = 0$ is established. (24)

Lines 59–65 are executed in order to bound the value of $\pi(j,v)$. We consider two cases.

- If $(\pi_{\max}(F) < c) \wedge (\pi(j,v) < c-1)$ holds before line 58, then by (23), $(\pi_{\max}(F) < c) \wedge (\pi(j,v) < c)$ holds after line 58. In this case, lines 60–65 need not be executed at all.

- If $\pi(j,v) = c-1$ holds before line 58, then $\pi(j,v) = c$ holds after line 58. In this case, we select some process r from AW^j_v (line 60) for erasing. If r is either an active process or a reserve process, then we may simply erase r (line 65). Otherwise, r was selected to cover some other variable u at some k^{th} step ($j < k \leq m_{H'}$), and was actually deployed at some l^{th} step ($k \leq l \leq m_{H'}$). In this case, we again apply the "exchange and erase" strategy (lines 62–65). (Since $\pi_{\max}(F) = c$ holds here, we can again use Lemma 1.) In either case, by (24), line 65 establishes $\pi(j,v) = 0$.

If r was a covering process (deployed or not) before executing line 65, then erasing r may in turn increase $\pi(ci(r), cv(r))$ by (23). If erasing r establishes $\pi(ci(r), cv(r)) = c$, then $\pi_{\max}(F) = c$ is again true after the

(a)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\{1..29, 56, s_1..s_{70}\}$	$\{30..38, 57, s_{71}..s_{140}\}$	$\cdots$	$\{39..46, r_1, r_4, s_{141}..s_{200}\}$	$\cdots$	$\{47..55, 58, 59, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^j \rvert$	100	80	$\cdots$	70	$\cdots$	90	$\cdots$
AW_v^j	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	80	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(j,v)$	0	0	$\cdots$	0	$\cdots$	0	$\cdots$

(b)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\boldsymbol{\{56, s_1..s_{70}\}}$ (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)	$\cdots$
$\lvert CW_v^j \rvert$	**71**	**71**	$\cdots$	**62**	$\cdots$	**81**	$\cdots$
AW_v^j	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	80	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(j,v)$	**9**	**9**	$\cdots$	**8**	$\cdots$	**9**	$\cdots$

(c)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\boldsymbol{\{s_1..s_{70}\}}$ (56 erased)	$\{57, s_{71}..s_{141}\}$	$\cdots$	$\{r_1, r_4, s_{141}..s_{200}\}$	$\cdots$	$\{58, 59, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^j \rvert$	**70**	71	$\cdots$	62	$\cdots$	81	$\cdots$
AW_v^j	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	80	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(j,v)$	**10**	9	$\cdots$	8	$\cdots$	9	$\cdots$

(d)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\{s_1..s_{70}\}$	$\{57, s_{71}..s_{140}\}$	$\cdots$	$\boldsymbol{\{r_4, s_{141}..s_{200}\}}$ (r_1 erased)	$\cdots$	$\{58, 59, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^j \rvert$	70	71	$\cdots$	**61**	$\cdots$	81	$\cdots$
AW_v^j	$\boldsymbol{\{r_2, r_3\}}$ (r_1 erased)	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	**70**	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(j,v)$	**0**	9	$\cdots$	**9**	$\cdots$	9	$\cdots$

Figure 17: An example of chain erasing. In this figure, we assume $c = 10$ and $m_F = 5$. Thus, we also have req$(j, v) = 10 \cdot |AW_v^j| + 50$. 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 52 sequentially. **(a)** Initial configuration before chain erasing starts. By assumption, $\pi(j, v) = 0$ holds for all covering pairs. **(b)** After erasing processes 1..55. No chain erasing is necessary so far. **(c)** After erasing 56 (line 58), we have $\pi(2, w) = 10 = c$. Thus, we find $\pi_{\max}(F) = c$ at line 59. Some process must be erased from AW_w^2. **(d)** We choose process r_1 at line 60, and erase it to reduce $\pi(2, w)$ to zero (line 65). 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 59, so no more chain erasing is necessary. (Continued on the next page.)

execution of line 65. Thus, we execute lines 60–65 again, and erase yet another process from $AW_{cv(r)}^{ci(r)}$. The chain erasing procedure continues in this manner as long as necessary.

Figure 17 shows two instances of chain erasing, as described above. 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).

(e)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\{s_1..s_{70}\}$	$\boldsymbol{\{s_{71}..s_{140}\}}$ (57 erased)	$\cdots$	$\{r_4, s_{141}..s_{200}\}$	$\cdots$	$\{58, 59, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^j\rvert$	70	**70**	$\cdots$	61	$\cdots$	81	$\cdots$
AW_v^j	$\{r_2, r_3\}$	$\{r_4, r_5, r_6\}$	$\cdots$	$\{59, r_7\}$	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	70	80	$\cdots$	70	$\cdots$	90	$\cdots$
$\pi(j,v)$	0	**10**	$\cdots$	9	$\cdots$	9	$\cdots$

(f)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\{s_1..s_{70}\}$	$\{s_{71}..s_{140}\}$	$\cdots$	$\boldsymbol{\{s_{141}..s_{200}\}}$ (r_4 erased)	$\cdots$	$\boldsymbol{\{58, s_{201}..s_{279}\}}$ (59 erased)	$\cdots$
$\lvert CW_v^j\rvert$	70	70	$\cdots$	**60**	$\cdots$	**80**	$\cdots$
AW_v^j	$\{r_2, r_3\}$	$\boldsymbol{\{r_5, r_6\}}$ (r_4 erased)	$\cdots$	$\boldsymbol{\{r_7\}}$ (59 erased)	$\cdots$	$\{r_8, r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	70	**70**	$\cdots$	**60**	$\cdots$	90	$\cdots$
$\pi(j,v)$	0	**0**	$\cdots$	**0**	$\cdots$	**10**	$\cdots$

(g)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\{s_1..s_{70}\}$	$\{s_{71}..s_{140}\}$	$\cdots$	$\{s_{141}..s_{200}\}$	$\cdots$	$\{58, s_{201}..s_{279}\}$	$\cdots$
$\lvert CW_v^j\rvert$	70	70	$\cdots$	60	$\cdots$	80	$\cdots$
AW_v^j	$\{r_2, r_3\}$	$\{r_5, r_6\}$	$\cdots$	$\{r_7\}$	$\cdots$	$\boldsymbol{\{r_9, r_{10}, r_{11}\}}$ (r_8 erased)	$\cdots$
req(j,v)	70	70	$\cdots$	60	$\cdots$	**80**	$\cdots$
$\pi(j,v)$	0	0	$\cdots$	0	$\cdots$	**0**	$\cdots$

(h)

covering pair (j,v)	$(2,w)$	$(2,x)$	$\cdots$	$(3,y)$	$\cdots$	$(5,z)$	$\cdots$
CW_v^j	$\{s_1..s_{70}\}$	$\{s_{71}..s_{140}\}$	$\cdots$	$\{s_{141}..s_{200}\}$	$\cdots$	$\boldsymbol{\{s_{201}..s_{279}\}}$ (58 erased)	$\cdots$
$\lvert CW_v^j\rvert$	70	70	$\cdots$	60	$\cdots$	**79**	$\cdots$
AW_v^j	$\{r_2, r_3\}$	$\{r_5, r_6\}$	$\cdots$	$\{r_7\}$	$\cdots$	$\{r_9, r_{10}, r_{11}\}$	$\cdots$
req(j,v)	70	70	$\cdots$	60	$\cdots$	80	$\cdots$
$\pi(j,v)$	0	0	$\cdots$	0	$\cdots$	**1**	$\cdots$

Figure 17: An example of chain erasing, continued. **(e)** After erasing 57 (line 58), 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 65), which in turn establishes $\pi(3,y) = c$. Hence, we execute lines 60–65 again, and erase 59 to lower $\pi(3,y)$. (This is an example of a process in *CE* being erased at line 65.) However, this in turn establishes $\pi(5,z) = c$. **(g)** We erase r_8 to reduce $\pi(5,z)$ to zero. Note that r_8 is an active process, since $r_8 \in AW_z^5 \subseteq \text{Act}^5(F) = \text{Act}(F)$. Thus, we have established $\pi_{\max}(F) < c$, and the inner loop (lines 59–65) terminates. (If m_F were bigger than 5, the inner loop might further continue, as long as necessary.) **(h)** Finally, we assign $p := 58$ at line 52 and erase 58. No further chain erasing is necessary.

Note that, in some cases, erasing a process $q \in CE$ results in erasing another process in $\text{Act}(F)$, if we choose a process $r \in \text{Act}(F)$ at line 60. If such a 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.

Fortunately, we can prove that the chain erasing procedure actually erases 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. (25)
- Erasing a process r *not* in *CE* decreases $\pi(F)$ by at least $c-1$. (26)

In order to prove (25), consider a covering pair (k, u). By definition, erasing q may change $\pi(k, u)$ only if either $q \in AW_u^k$ or $q \in CW_u^k$ holds. If $q \in AW_u^k$ holds, then by (12), $\mathrm{req}(k, u)$ is decreased by c, and hence $\pi(k, u)$ cannot increase. On the other hand, if $q \in CW_u^k$ holds, then by (13), $\pi(k, u)$ may increase by at most one. Since $q \in CW_u^k$ holds for at most one covering pair (k, u) (namely, $(ci(q), cv(q))$), it follows that erasing q may increase $\pi(F)$ by at most one.

We now prove (26). Note that a process $r \notin CE$ may be erased only at line 65. In this case, we have $r \in AW_v^j$ and $\pi(j, v) = c$, and by (24), $\pi(j, v)$ is reduced to zero. (For example, in Figure 17(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 (l, w) is changed as follows:

(i) if $(l, w) = (j, v)$, then $\pi(l, w)$ decreases by c;

(ii) otherwise, if $r \in AW_w^l$, then by (12) and (13), $\pi(l, w)$ cannot increase;

(iii) otherwise, if $r \in CW_w^l$, then by (13), $\pi(l, w)$ increases by at most one;

(iv) otherwise, $\pi(l, w)$ does not change.

Since $r \in CW_w^l$ holds for at most one covering pair (l, w), Case (iii) may apply to at most one covering pair. By definition (given in (15)), $\pi(F)$ is the sum of the ranks of all covering pairs. Therefore, by summing over Cases (i)–(iv), we have (26).

By (25), it follows that $\pi(F)$, being initially zero, increases by at most $|CE|$ throughout the execution of ChainErase. Since $\pi(F)$ is always nonnegative by definition, by (26), processes *not* in CE may be erased at most $|CE|/(c-1)$ times. Since $CE \subseteq \mathrm{Cvr}(H')$ (line 17), it follows that we erase at most $|CE|/(c-1)$ active processes in total.

The argument explained so far (and proved formally in Appendix B) establishes the following theorem.

Theorem 3 *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 $N = |P|$.* □

5 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 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.

Our work suggests 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 $\Omega(\log N)$ is a tight lower bound on the number of distinct variables remotely accessed, even when restricting attention to single-process executions. Another interesting question arises from our lower-bound proof. This 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.

Acknowledgement: We are grateful to the anonymous referees for their helpful suggestions concerning an earlier draft of this paper.

References

[1] 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.

[2] 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.

[3] 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.

[4] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[5] R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation*, 126(1):62–73, April 1996.

[6] J. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, May 1993.

[7] J. Anderson and M. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28(2):99–103, June 1988.

[8] 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.

[9] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, January 2001.

[10] 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.

[11] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2003.

[12] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[13] 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.

[14] 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.

[15] 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.

[16] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[17] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[18] 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.

[19] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, 1995.

[20] J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.

[21] 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.

[22] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[23] L. Lamport. The mutual exclusion problem: Part II - Statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[24] L. Lamport. On interprocess communication: Part II - Algorithms. *Distributed Computing*, 1:86–101, 1986.

[25] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[26] L. Lamport. win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, July 1990.

[27] 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.

[28] G. Peterson and J. Burns. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual ACM Symposium on the Foundation of Computer Science*. ACM, 1987.

[29] R. Schaffer. On the correctness of atomic multi-writer registers. Technical Report MIT/LCS/TM-364, Laboratory for Computer Science, MIT, Cambridge, 1988.

[30] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, 1994.

[31] 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.

[32] 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.

[33] P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

[34] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

Appendix A: Detailed Correctness Proof of Algorithm NA

In this appendix, we present a detailed correctness proof of Algorithm NA. To simplify the proof, we present a recursive version of Algorithm NA, shown in Figure 18. In particular, only the root node (lines 2–20) is presented in detail, and the details of the left and right subtrees beneath the root are hidden in lines 1 and 21. We begin by stating several notational conventions that will be used in our proof.

We use $s.p$ to denote the statement with label s of process p, and $p.v$ 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. As shown in [7], a nonatomic algorithm can be converted into an equivalent atomic algorithm as follows: we assume that all reads execute atomically, and replace each nonatomic write $v := val$ (where val is an expression over private variables) by the following code fragment.[14]

```
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
```

For example, in Figure 18, if process p executes line 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)** $R2[p] = true \wedge p@\{6\}$. (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.) Lines 7, 11, and 20 also have similar properties. Note that these four lines are the only writes of nonatomic variables in Algorithm NA, since variables T and C are implemented using register constructions. Variables $Q1$ and $R1$ cannot be so implemented, because then we would have writes inside the busy-waiting loops of lines 12, 13, and 15. Variables $Q2$ and $R2$ can be implemented either way; in this proof, we assume that they are nonatomic. As a result, we assume that each labeled sequence of statements (except lines 5, 7, 11, and 20) in Figure 18 is atomic. For example, statement $8.p$ reads $C[1-p.side]$, stores its value to p's private variable $p.rival$, and establishes either $p@\{9\}$ (if $C[1-p.side] \neq \perp$ holds) or $p@\{16\}$ (otherwise). We number statements in this way to reduce the number of cases that must be considered in the proof. Note that each numbered sequence of statements (except lines 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 (lines 1 and 21) are correct mutual exclusion algorithms, and hence we may assume that lines 1 and 21 execute atomically. Moreover, we have the following invariant.

invariant $\big|\{p :: p@\{2..21\} \wedge p.side = s\}\big| \leq 1$ (I0)

Invariant (I0) states that at most one process may execute lines 2–21 from either side at any time. Thus, we need to consider at most two processes executing in these lines at any given state.

We now prove that each of invariants (I1)–(I16), stated below, is an invariant. Invariant (I6) establishes the mutual exclusion property; invariants (I7)–(I16) are used to prove starvation freedom. (Many of these invariants are adapted from [34].) 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.

[14] This code transformation is purely for the ease of correctness proof; we still consider the execution of this whole code fragment as a single RMR operation.

```
shared variables                                        private variables
    T: 0..N − 1;                                            qtoggle, rtoggle, temp: boolean;
    C: array[0,1] of (0..N − 1, ⊥) initially ⊥;             rival: 0..N − 1, ⊥
    Q1, Q2, R1, R2: array[0..N − 1] of boolean
                                                        private constant
                                                            side = if p < N/2 then 0 else 1 fi

process p ::   /* 0 ≤ 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 := ¬R1[p];
5:  R2[p] := rtoggle;
6:  qtoggle := ¬Q1[p];
7:  Q2[p] := qtoggle;
8:  rival := C[1 − side];
    if (rival ≠ ⊥ ∧
9:            T = p) then
10:       temp := Q2[rival];
11:       Q1[rival] := temp;
12:       await (Q1[p] = qtoggle) ∨
13:                 (R1[p] = rtoggle);
14:       if T = p then
15:           await R1[p] = rtoggle fi
    fi;
16: Critical Section;

17: C[side] := ⊥;
18: rival := T;
    if rival ≠ 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 18: Recursive version of ALGORITHM NA. Only lines 5, 7, 11, and 20 are executed nonatomically.

Invariant (I1) states that variable $C[0]$ ($C[1]$) accurately represents the winner of the left (right) subtree. (For the reader's convenience, we provide several "primed" invariants that are created by exchanging s with $1-s$ and p with q. Needless to say, it suffices to prove only the "unprimed" versions.)

$$\textbf{invariant} \quad \big(C[s] = p \ \wedge \ p \neq \bot\big) = \big(p@\{3..17\} \ \wedge \ p.side = s\big) \qquad \text{(I1)}$$
$$\textbf{invariant} \quad \big(C[1-s] = q \ \wedge \ q \neq \bot\big) = \big(q@\{3..17\} \ \wedge \ q.side = 1-s\big) \qquad \text{(I1}'\text{)}$$

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 (I2) states that, if two processes p and q are competing (at the root node), then variable T holds the identity of one of them. Hence, T can be used as a tie-breaker.

$$\textbf{invariant} \quad p@\{4..20\} \ \wedge \ q@\{4..20\} \ \wedge \ p \neq q \ \Rightarrow \ T = p \ \vee \ T = q \qquad \text{(I2)}$$

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 (I3) states that, if a process p has read the value of $C[1-p.side]$ at line 8, then either p correctly knows the identity of its rival, q (*i.e.*, $p.rival = q$), or q has executed line 3 after p did (*i.e.*, $T = q$). In the latter case, ties are broken in favor of p, so p need not know the identity of q.

invariant $p@\{9..17\} \wedge q@\{4..17\} \wedge p \neq q \Rightarrow p.rival = q \vee T = q$ (I3)
invariant $q@\{9..17\} \wedge p@\{4..17\} \wedge p \neq q \Rightarrow q.rival = p \vee T = p$ (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 lines 2–21.) □

Recall from Section 2 that statements 4.p and 5.p seek to establish $R1[p] \neq R2[p]$, while statements 19.q and 20.q (where q is p's rival) seek to establish $R1[p] = R2[p]$. (The latter condition tells p that it can enter its critical section.) Invariant (I4) states that, if the value of $R1[p]$ has changed after the execution of 4.p, then ties are broken in favor of p, so p can safely enter its critical section.

invariant $p@\{5..15\} \wedge q@\{4..19\} \wedge R1[p] = p.rtoggle \wedge p \neq q \Rightarrow T = q$ (I4)
invariant $q@\{5..15\} \wedge p@\{4..19\} \wedge R1[q] = q.rtoggle \wedge p \neq q \Rightarrow T = p$ (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. □

Invariant (I5) implies that, if p and q are competing, and if p executes its critical section (line 16), then ties are indeed broken in favor of p.

invariant $p@\{16..20\} \wedge q@\{4..17\} \wedge p \neq q \Rightarrow T = q$ (I5)
invariant $q@\{16..20\} \wedge p@\{4..17\} \wedge p \neq q \Rightarrow T = p$ (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). □

invariant (Mutual Exclusion) $\left|\{p :: p@\{16\}\}\right| \leq 1$ (I6)

Proof: For the sake of contradiction, assume that there are two distinct processes, p and q, such that $p@\{16\} \wedge q@\{16\}$ holds. By (I5) and (I5′), we then have $T = q$ *and* $T = p$, a contradiction. □

Invariants (I7) and (I8) follow trivially from the structure of the algorithm.

invariant $p@\{6..15\} \Rightarrow R2[p] = p.rtoggle$ (I7)
invariant $p@\{8..15\} \Rightarrow Q2[p] = p.qtoggle$ (I8)

Invariant (I9) is similar to (I3); it states that, if a process p is busy-waiting in its entry section, and if its rival q has executed statement $18.q$, then q has correctly read the identity of p from T.

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 $18.q$. 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 $q.rival = p$.

Statement $8.p$ may establish the antecedent only if executed when $q@\{19, 20\}$ holds. By (I0), $p@\{8\} \wedge q@\{19, 20\}$ implies $p.side = 1 - q.side$.

We claim that $C[q.side] = \perp$ holds in this case. Assume otherwise. Then, we have $C[q.side] = r$, where r is any arbitrary process. Thus, by applying (I1) with 'p' $\leftarrow r$ and 's' $\leftarrow q.side$, we have $r@\{3..17\} \wedge r.side = q.side$. Taken together with $q@\{19, 20\}$, we have $r \neq q$, and hence we have a contradiction of (I0).

Therefore, we have $C[q.side] = C[1 - p.side] = \perp$. Thus, in this case, $8.p$ cannot establish $p@\{9..15\}$. □

Taken together with (I9), invariant (I10) states that statement $20.q$ eventually establishes $R1[p] = R2[p]$, which signals p to proceed to its critical section.

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. □

Invariant (I11) states that, if p is busy-waiting in its entry section, and if $R1[p] \neq R2[p]$ holds,[15] then there exists some rival process q that is still active. Hence, q will eventually execute statement $20.q$, which establishes $R1[p] = R2[p]$ as shown above.

invariant $p@\{9..15\} \wedge R1[p] = \neg p.rtoggle \Rightarrow (\exists q : q \neq p :: q@\{3..20\})$ (I11)

Proof: The only statements that may falsify (I11) are $4.p$ (by updating $p.rtoggle$), $8.p$ (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 $C[1 - p.side] = q \neq \perp$ 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. By (I5$'$), this implies $T = p$, 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 $R1[p] = p.rtoggle$ by (I7), (I9), and (I10). □

Invariants (I12) and (I13) imply that, if $p@\{10..15\} \wedge q@\{11..15\}$ holds, then one of the following holds: **(i)** ties are broken in favor of q (*i.e.*, $T = p$), or (if $T = q$) **(ii)** q will eventually establish (or has already established) $Q1[p] = Q2[p]$ by executing $11.q$. (Note that the antecedent of (I12) or (I13) implies $q.rival = p$, by (I3$'$), and $Q2[p] = p.qtoggle$, by (I8).) In the former case, q will enter its critical section first. In the latter case, statement $11.q$ will signal p to proceed to line 14; since $T = q$, p will then immediately proceed to its critical section before q does. (This description does not cover the case of $(p@\{15\} \wedge T = q)$, which is dealt with later by (I16).)

[15] Note that the antecedent of (I11) implies $R1[p] \neq R2[p]$, by (I7).

invariant $p@\{10..15\} \wedge q@\{11\} \wedge p \neq q \Rightarrow q.temp = Q2[p] \vee T = p$ (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 (I3′), we have $q.rival = p \vee T = p$. Moreover, if $q.rival = p$ holds, then 10.q establishes $q.temp = Q2[p]$.

The only statements that may falsify the consequent are 7.p (by updating $Q2[p]$), 10.q and 19.q (by updating $q.temp$), 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 $p@\{10..15\} \wedge q@\{12..15\} \wedge p \neq q \Rightarrow Q1[p] = p.qtoggle \vee T = p$ (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′), (I8), and (I12), we have $q.rival = p \vee T = p$, $Q2[p] = p.qtoggle$, and $q.temp = Q2[p] \vee T = p$, 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. □

By invariant (I14), if both p and q are busy-waiting at lines 12–13, then at least one of them eventually stops waiting.

invariant $p@\{12, 13\} \wedge q@\{12, 13\} \wedge p \neq q \Rightarrow Q1[p] = p.qtoggle \vee Q1[q] = q.qtoggle$ (I14)

Proof: Since $T \neq p \vee T \neq q$ is always true, (I14) follows easily from (I13). □

Invariant (I16), given later, implies that if a process p is busy-waiting at line 15, and if ties are broken in favor of p, then p eventually enters its critical section. The following invariant is used as an intermediate step toward the proof of (I16); it enumerates all the possible locations in which p's rival (q) might be executing, together with the possible values of $R1[p]$ and $R2[p]$.

invariant

$$
\begin{array}{rll}
p@\{7..15\} \Rightarrow & p@\{7..13\} \wedge Q1[p] = \neg p.qtoggle \wedge R1[p] = \neg p.rtoggle & \text{(D1)} \\
\vee & \big(\exists q : q \neq p :: q@\{11..18\}\big) \wedge R1[p] = \neg p.rtoggle & \text{(D2)} \\
\vee & \big(\exists q : q \neq p :: q@\{19\} \wedge q.rival = p\big) \wedge R1[p] = \neg p.rtoggle & \text{(D3)} \\
\vee & \big(\exists q : q \neq p :: q@\{20\} \wedge q.rival = p \wedge q.temp = R2[p]\big) & \text{(D4)} \\
\vee & \big(\exists q : q \neq p :: q@\{2, 3, 21\}\big) \wedge R1[p] = p.rtoggle & \text{(D5)} \\
\vee & \neg\big(\exists q : q \neq p :: q@\{2..21\}\big) \wedge R1[p] = p.rtoggle & \text{(D6)} \\
\vee & p@\{7..13\} \wedge \big(\exists q : q \neq p :: q@\{20\}\big) \wedge Q1[p] = \neg p.qtoggle & \text{(D7)} \\
\vee & T \neq p & \text{(D8)}
\end{array}
\qquad \text{(I15)}
$$

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 $C[1 - q.side] = p$. Also, by (I5′), we have $T = p$. Thus, in this case, $18.q$ establishes (D3). By (I0), $20.r$ cannot be executed while $p@\{7..15\} \wedge q@\{11..18\}$ 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 $p@\{7..15\} \wedge q@\{19\}$ 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 $q@\{21\} \wedge R1[p] = R2[p]$. 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 $p@\{7..15\} \wedge q@\{2, 3, 21\}$ 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. □

invariant $p@\{15\} \wedge T = q \wedge p \neq q \Rightarrow R1[p] = p.rtoggle \wedge q@\{4..15\}$ (I16)
invariant $q@\{15\} \wedge T = p \wedge p \neq q \Rightarrow R1[q] = q.rtoggle \wedge p@\{4..15\}$ (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 $p@\{15\} \wedge T = q \wedge R1[q] = q.rtoggle$ holds, which is precluded by (I4′). By (I0), statement $20.r$ cannot be executed while $p@\{15\} \wedge q@\{4..15\}$ holds. □

We now prove the following "unless" properties. (A *unless* B is true if the following holds: if A holds before some statement execution, then $A \vee B$ holds after that execution. Informally, A is not falsified until B is established.) Taken together, (U1)–(U4) imply the following: *Assume that* $p@\{12..15\} \wedge R1[p] = p.rtoggle$ *holds. If* q *is* ***not*** *expected to enter its critical section before* q *(i.e.,* $q@\{4..15\} \wedge T = p$ *is false), then* $R1[p] = p.rtoggle$ *continues to hold, allowing* p *to proceed to its critical section.*

$$p@\{12..15\} \wedge R1[p] = p.rtoggle \wedge (\exists q : q \neq p :: q@\{21\}) \quad \textit{unless} \quad p@\{16\} \vee \neg(\exists q : q \neq p :: q@\{2..21\}) \tag{U1}$$

$$p@\{12..15\} \wedge R1[p] = p.rtoggle \wedge \neg(\exists q : q \neq p :: q@\{2..21\}) \quad \textit{unless} \quad p@\{16\} \vee (\exists q : q \neq p :: q@\{2, 3\}) \tag{U2}$$

$$p@\{12..15\} \wedge R1[p] = p.rtoggle \wedge q@\{2, 3\} \quad \textit{unless} \quad p@\{16\} \vee (q@\{4..15\} \wedge T = q) \tag{U3}$$

$$p@\{12..15\} \wedge R1[p] = p.rtoggle \wedge q@\{4..15\} \wedge T = q \quad \textit{unless} \quad p@\{16\} \tag{U4}$$

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 (I4′), the left-hand side of (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. □

Proof of starvation freedom. We begin by proving livelock freedom. It suffices to consider the loops represented by the **await** statements at lines 12–13 and line 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 (I16′), 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 $T = q$ or $T = p$. Thus, by (I16) and (I16′), we have either $R1[p] = p.rtoggle$ or $R1[q] = q.rtoggle$, and hence either p or q eventually terminates its **await** statement.[16]

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 lines 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 line 18. By (I5′), q finds $T = p$ at line 18, and hence it executes lines 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.

[16] In fact, with several more invariants, it can be shown that $p@\{15\} \wedge q@\{15\}$ is impossible.

Appendix B: Detailed Lower-bound Proof

In this appendix, our lower-bound proof is presented in detail. 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 6). (27)
- For each m and v, $\mathrm{Act}(H)$ and $CW_v^m(H)$ are disjoint. (28)
- For each m and v, $\mathrm{Act}(H)$ and $RW_v^m(H)$ are disjoint. (29)
- Each invocation event e_p in H is the last event in $H \,|\, p$. (30)
- Each atomic write or read event f_q in H is contained in some active segment $S(q, m)$. (31)

We now present several lemmas. Throughout this appendix, we assume the existence of a fixed one-shot mutual exclusion system $\mathcal{S} = (C, P, V)$. Lemma 2 asserts that information flow does not happen in a regular computation.

Lemma 2 *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:*

$$\mathit{last_writer}(v, F) = p \;\vee\; \mathit{last_writer}(v, F) = \perp \;\vee\; \mathit{value}(v, F) = \star.$$

Proof: Let $f_q = \mathit{last_writer_event}(v, F)$. If we have either $q = p$ or $q = \perp$, then we are done. Thus, assume $q \neq p \;\wedge\; q \neq \perp$. 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 (31),

- f_q is contained in solo segment $S(q, m)$, for some m. (32)

Thus,

- $ce(q, j)$ is a write to v, for some $j \leq m$; (33)
- $q \in AW_v^j$. (34)

By (32), H contains $S(q, m)$. Thus, by (8),

$$q \in \mathrm{Act}^m(H). \tag{35}$$

Since e_p reads v, by (31), 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 $l \geq m$. (36)

Therefore, from the structure of H^l, given in (8), we have $p \in \mathrm{Act}^l(H)$, and hence, since $\mathrm{Act}^l(H) \subseteq \mathrm{Act}^m(H)$ (by (6) and (7)), we also have

$$p \in \mathrm{Act}^m(H). \tag{37}$$

Also, since e_p reads v,

- $ce(p, k)$ is a read of v, for some $k \leq l$. (38)

We consider two cases. (Note that j is defined in (33).)

First, if CW_v^j is nonempty, then by R2, (33), and (35), $C(q, m)$ contains an invocation event g on v. Thus, by (32) and (36), 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 $f_q = \mathit{last_writer_event}(v, F)$.

Second, if CW_v^j is empty, then by (34), (38), 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 (33)), we also have $\mathrm{Cvr}^j(H) \subseteq \mathrm{Cvr}^m(H)$ (by (6)), and hence $p \in \mathrm{Cvr}^m(H)$. However, since $\mathrm{Cvr}^m(H)$ and $\mathrm{Act}^m(H)$ are disjoint (by (7)), we have a contradiction of (37). □

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 the write of some variable v by some process r, and hence we cannot apply erase_p directly, since that may cause r's write to v to be "uncovered" and create information flow. In that case, we first apply operator $\mathit{exchange}_{pq}$. Informally, $\mathit{exchange}_{pq}$ 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). Thus, p becomes a reserve process in $\mathit{exchange}_{pq}(H)$, and hence can be safely erased by applying erase_p. (This "erase after exchange" strategy is formally described in Lemma 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 8; formally, this property follows from (4), (6), (7), (8), (10).)

We define $\mathit{erase}_p(H)$ to be the computation where these segments are erased, *i.e.*, $\mathit{erase}_p(H) = H - (S(p,1) \circ C(p,1)) - (S(p,2) \circ C(p,2)) - \cdots - (S(p,m') \circ C(p,m'))$, where m' is defined to be m_H (if p is active) or $m-1$ (if $p \in RW_v^m(H)$ holds). □

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 $ie(p,m)$ with $ie(q,m)$, *i.e.*, $\mathit{exchange}_{pq}(H) = F \circ \langle ie(q,m) \rangle \circ G$. □

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 $cp(q) = r$, 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 3 and 4.

Lemma 3 *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 ($1 \leq j \leq m_H$), k ($1 \leq k \leq m_H$), variable w, and process s:*

$$P(H') = P(H); \tag{39}$$
$$\mathrm{Act}^j(H') = \mathrm{Act}^j(H); \tag{40}$$
$$\mathrm{Cvr}^j(H') = \mathrm{Cvr}^j(H); \tag{41}$$
$$AW_w^j(H') = AW_w^j(H); \tag{42}$$
$$CW_w^j(H') = CW_w^j(H); \tag{43}$$

$$RW_w^j(H') = \begin{cases} (RW_w^j(H) \cup \{p\}) - \{q\}, & \text{if } j = m \text{ and } w = v \\ RW_w^j(H), & \text{otherwise.} \end{cases} \tag{44}$$

Proof: Note that, by the definition of $exchange_{pq}$, the only difference between H and H' is that $ie(p, m)$ is replaced by $ie(q, m)$. Moreover, both $ie(p, m)$ and $ie(q, m)$ 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 (39)–(44) follow immediately from the definition of $exchange_{pq}$. Since H satisfies (2)–(10) and R1–R4, by applying (39)–(44), assertions (2)–(10) and R1–R4 follow immediately. It follows that H' is also regular. □

Lemma 4 *Consider a regular computation H in C with induction number m_H, and a process p. Assume either of the following:*

- $p \in \mathrm{Act}(H) \ \wedge \ |\mathrm{Act}(H)| \geq 2$, *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 $H' = erase_p(H)$. 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:

$$\begin{aligned} P(H') &= P(H) - \{p\}; && (45) \\ \mathrm{Act}^j(H') &= \mathrm{Act}^j(H) - \{p\}; && (46) \\ \mathrm{Cvr}^j(H') &= \mathrm{Cvr}^j(H) - \{p\}; && (47) \\ AW_w^j(H') &= AW_w^j(H) - \{p\}; && (48) \\ CW_w^j(H') &= CW_w^j(H) - \{p\}; && (49) \\ RW_w^j(H') &\supseteq RW_w^j(H). && (50) \end{aligned}$$

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 (30), each such removed event (except events by p) is the last event by that process. Therefore,

- for each process q such that $H' \,|\, q \neq \langle\rangle$, $H' \,|\, q$ is either $H \,|\, q$, or the subcomputation of $H \,|\, q$ obtained by removing its last event. (51)

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

$$H = F \circ \langle e_q \rangle \circ G \quad \text{and} \quad H' = F' \circ \langle e_q \rangle \circ G', \tag{52}$$

where F and G (respectively, F' and G') are subcomputations of H (respectively, H'). Also, by the definition of $erase_p$, and by (51), we have the following:

- F' is a subcomputation of F; (53)
- $F' \,|\, q = F \,|\, q$. (54)

By induction, we have

$$F' \in C. \tag{55}$$

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 $\mathit{last_writer_event}(u, F') \neq \bot$, then $\mathit{value}(u, F') = \mathit{value}(u, F)$ holds.

Proof of Claim: By (31),

- e_q is contained in some solo segment $S(q, l)$. (56)

Hence, since e_q reads u,

- $ce(q, j)$ reads u, for some $j \leq l$. (57)

Also, by (8) and (56), we have

$$q \in \mathrm{Act}^l(H). \tag{58}$$

Note that, by (53), $\mathit{last_writer_event}(u, F') \neq \bot$ implies

$$\mathit{last_writer}(u, F) \neq \bot. \tag{59}$$

We consider two cases. First, assume that $\mathit{last_writer}(u, F) = q$ holds. Let $f_q = \mathit{last_writer_event}(u, F)$. By (54), f_q is also contained in F'. By (53), this implies that $f_q = \mathit{last_writer_event}(u, F')$, and hence the claim follows.

Second, assume that $\mathit{last_writer}(u, F) \neq q$ holds. In this case, by (59), and by applying Lemma 2 with 'e_p' $\leftarrow e_q$ and 'v' $\leftarrow u$, we have $\mathit{value}(u, F) = \star$. (The assumptions stated in Lemma 2 follow from (52) 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

$$f_r = \mathit{last_writer_event}(u, F'). \tag{60}$$

(By the assumption of Claim 1, $f_r \neq \bot$ holds.) Since $\mathit{last_writer}(u, F) \neq q$, 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 (31),

- f_r is contained in some solo segment $S(r, l')$. (61)

Hence, by (8),

$$r \in \mathrm{Act}^{l'}(H). \tag{62}$$

Since f_r is contained in F', by (52), (53), and the definition of erase_p,

- f_r precedes e_q in H, and (63)
- $r \neq p$. (64)

Combining (56), (61), and (63), we also have $l' \leq l$. Since f_r writes u,

- $ce(r, k)$ is a write to u for some $k \leq l'$. (65)

We claim that $CW_u^k(H)$ is nonempty. Assume otherwise. Then, by (57) and (65), 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 (65)), we also have $\mathrm{Cvr}^k(H) \subseteq \mathrm{Cvr}^l(H)$ (by (6)), 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)), we have a contradiction of (58).

It follows that $CW_u^k(H)$ is nonempty, and hence, by (62) and (65), 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 (56), (61), and (63), 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 (64), 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 (60). $\square$

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 (55), 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 $last_writer_event(u, F') \neq \bot$ holds. By Claim 1, we again have $value(u, F') = value(u, F)$. Thus, by (55), 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 $last_writer_event(u, F') = \bot$ holds. By (31), e_q is contained in some solo segment $S(q, m)$. Hence, by the definition of a solo segment, $(F' \,|\, q) \circ \langle e_q \rangle$ is a valid solo computation, that is,

$$(F' \,|\, q) \circ \langle e_q \rangle \in C. \tag{66}$$

Since $F' \,|\, q$ is a subcomputation of F', we also have

$$value(u, F' \,|\, q) = value(u, F') = (\text{initial value of } u). \tag{67}$$

By (55), (66), (67), and applying P2 with 'H' $\leftarrow F' \,|\, q$, 'e_p' $\leftarrow e_q$, 'G' $\leftarrow F'$, and 'v' $\leftarrow u$, the claim follows. □

Step 2. We now show that H' is regular, and that H' satisfies (45)–(50). Assertions (45)–(50) follow immediately from the definition of $erase_p$. Since H satisfies (2)–(10), by applying (45)–(50), it can be easily shown that H' also satisfies (2)–(10).

In order to show that H' has induction number m_H, it suffices to prove that $(H')^{m_H}$ is not an empty computation. By (8), this is equivalent to saying that $\mathrm{Act}^{m_H}(H')$ is nonempty. By applying (46) with 'j' $\leftarrow m_H$, we have

$$\mathrm{Act}^{m_H}(H') = \mathrm{Act}^{m_H}(H) - \{p\} = \mathrm{Act}(H) - \{p\}.$$

If Condition (A) is true, then this clearly implies $\mathrm{Act}^{m_H}(H') \neq \{\}$; if Condition (B) is true, then by (29), we have $\mathrm{Act}^{m_H}(H') = \mathrm{Act}(H) \neq \{\}$. It follows that H' has induction number m_H.

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: $e_r = ie(q, j)$, $q \in CW^j_u(H)$, $cp(q; H) = r$, and $ce(r, j)$ is a write to u. By (49) and $q \neq p$, we have $q \in CW^j_u(H')$. By $r \neq p$, and by the definition of $erase_p$, $cp(q; H') = r$ 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 $ce(q, j)$ writes some variable u and $CW^j_u(H')$ is nonempty. By (46), we have $q \in \mathrm{Act}^l(H)$ and $q \neq p$. By (49), $CW^j_u(H)$ 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 $ie(r, j)$ for some $r \in CW^j_u(H)$. 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 (49), we have $r \in CW^j_u(H')$. It follows that $C(q, l; H')$ contains exactly one invocation event on u (namely, $ie(r, j)$), where $r \in CW^j_u(H')$. 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 $AW^j_w(H)$ and $CW^j_w(H)$ are nonempty, then so is $CW^j_w(H')$.

Proof of Claim: We consider two cases. First, if Condition (A) holds, then since $\mathrm{Act}(H)$ and $CW^j_w(H)$ are disjoint (by (28)), we have $p \notin CW^j_w(H)$. Therefore, by (49), $CW^j_w(H')$ equals $CW^j_w(H)$, and hence is nonempty.

Second, assume that Condition (B) holds. If $(j, w) = (m, v)$, then since $AW^j_w(H)$ is nonempty, we have $|CW^m_v(H)| \geq 2$, and hence, by (49), $CW^j_w(H')$ $(= CW^m_v(H'))$ is nonempty. On the other hand, if $(j, w) \neq (m, v)$, then $CW^j_w(H)$ and $CW^m_v(H)$ are disjoint by (4). Moreover, by Condition (B), we have $p \in RW^m_v(H) \subseteq CW^m_v(H)$. Thus we have $p \notin CW^j_w(H)$. Therefore, by (49), we have $CW^j_w(H') = CW^j_w(H)$, and hence $CW^j_w(H')$ is nonempty. □

We now claim that H' satisfies R4. Consider some segment index l, process q, and variable u such that $q \in AW^l_u(H')$ and $CW^l_u(H')$ is empty. By (48), we have $q \in AW^l_u(H)$ and $q \neq p$. By Claim 3, $CW^l_u(H)$ 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 $CW^j_u(H)$ 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 (46), $r \in \mathrm{Act}^j(H')$ implies $r \in \mathrm{Act}^j(H)$ and $r \neq p$. Note that, if (i) is true, then $r \in AW^j_u(H)$ holds by definition. Thus, by Claim 3, it follows that $CW^j_u(H')$ is also nonempty. Also note that, since $r \neq p$, if (ii) is true, then $r \in \mathrm{Cvr}^l(H)$ and (47) imply $r \in \mathrm{Cvr}^l(H')$. From these assertions, R4 easily follows. □

The next lemma is a simple application of Lemma 4. It states that we can safely erase any proper subset of active processes.

Lemma 5 *Consider a regular computation H in C with induction number m_H, and a **proper** 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:

- $\pi_{\max}(H') \leq \pi_{\max}(H)$; (68)
- *for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q,*

$$\begin{aligned} P(H') &= P(H) - K; & (69)\\ \mathrm{Act}^j(H') &= \mathrm{Act}^j(H) - K; & (70)\\ \mathrm{Cvr}^j(H') &= \mathrm{Cvr}^j(H); & (71)\\ AW^j_w(H') &= AW^j_w(H) - K; & (72)\\ CW^j_w(H') &= CW^j_w(H); & (73)\\ RW^j_w(H') &\supseteq RW^j_w(H). & (74) \end{aligned}$$

Proof: By inductively applying Lemma 4, assertions (69)–(74) follow easily. (Note that, after applying the *erase* operator i times $(0 \leq i < h)$, we have $|\mathrm{Act}(H)| - i$ active processes left, by (46). Since $|K| = h < |\mathrm{Act}(H)|$, this implies Condition (A) of Lemma 4.) As for (73), note that (28) implies that $\mathrm{Act}(H)$ and $CW^j_w(H)$ are disjoint for each j and w. Since $K \subseteq \mathrm{Act}(H)$, this also implies that K and each $CW^j_v(H)$ are disjoint, and hence (73) follows. Similarly, by (6) and (28), $\mathrm{Act}(H)$ and $\mathrm{Cvr}^j(H)$ are disjoint for each j, from which (71) follows.

We now prove that H' satisfies (68). 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

$$AW^j_w(H') \neq \{\} \ \wedge \ CW^j_w(H') \neq \{\}. \quad (75)$$

By (72), we have

$$|AW_w^j(H')| \le |AW_w^j(H)|. \qquad (76)$$

Thus, since both H and H' have induction number m_H, by the definition of 'req', given in (12), we have

$$\mathrm{req}(j, w; H') \le \mathrm{req}(j, w; H). \qquad (77)$$

By (73), (75), and (76), it follows that (j, w) is also a covering pair in H. Thus, by (13), we have $\pi(j, w; H) = \max\{0,\ \mathrm{req}(j, w; H) - |CW_w^j(H)|\}$. Combining this with (73) and (77), assertion (68) follows. □

The next lemma is a simple application of Lemmas 3 and 4. Given a regular computation H satisfying $\pi_{\max}(H) \le c$, 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 $\pi_{\max}(H) \le c$ 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 rank, by reducing the number of covering processes.)

Lemma 6 *Consider a regular computation H in C with induction number m_H, and a process p. Assume the following:*

- $m_H \le c - 2$, (78)
- $\pi_{\max}(H) \le c$, (79)
- $p \in P(H)$, *and* (80)
- $|\mathrm{Act}(H)| \ge 2$. (81)

Then, there exists a regular computation H' in C with induction number m_H, satisfying the following for each segment index j ($1 \le j \le m_H$) 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;* (82)
- *if $p \in CW_w^j(H)$, then $\pi(j, w; H') \le \pi(j, w; H) + 1$;* (83)
- *if $p \notin CW_w^j(H)$, then $\pi(j, w; H') \le \pi(j, w; H)$;* (84)
- *the following hold:*

$$\begin{aligned} P(H') &= P(H) - \{p\}; & (85)\\ \mathrm{Act}^j(H') &= \mathrm{Act}^j(H) - \{p\}; & (86)\\ \mathrm{Cvr}^j(H') &= \mathrm{Cvr}^j(H) - \{p\}; & (87)\\ AW_w^j(H') &= AW_w^j(H) - \{p\}; & (88)\\ CW_w^j(H') &= CW_w^j(H) - \{p\}. & (89) \end{aligned}$$

Proof: First, we consider the case in which $p \in \mathrm{Act}(H)$ holds. Let $H' = erase_p(H)$. By applying Lemma 4, assertions (85)–(89) can be easily shown to be true. (Condition (A) of Lemma 4 follows from (81).) Moreover, for each segment index j and variable w, $p \in \mathrm{Act}(H)$ implies $p \notin CW_w^j(H)$ (by (3) and (4)). Therefore, by (89), we have $CW_w^j(H') = CW_w^j(H)$, and hence (82) follows. Combining $CW_w^j(H') = CW_w^j(H)$ with (12), (13), and (88), we also have $\pi(j, w; H') \le \pi(j, w; H)$, and hence we also have (83) and (84).

Thus, in the rest of the proof, we may assume $p \notin \mathrm{Act}(H)$. In this case, by (3), (4), and (80), we have the following:

$$p \in CW_v^m(H), \quad \text{for some } m\ (1 \le m \le m_H) \text{ and variable } v. \qquad (90)$$

We now establish the following simple claim.

Claim 1: $AW_v^m(H) = \{\} \ \vee\ |RW_v^m(H)| \ge 2$.

Proof of Claim: If $AW_v^m(H)$ is empty, then we are done. Otherwise, by (90), (m, v) is a covering pair in H. Thus, by applying Lemma 1, we have $|RW_v^m(H)| > |AW_v^m(H)| \geq 1$, and hence the claim follows. (Assumptions (16) and (17) stated in Lemma 1 follow from (78) and (79), respectively.) □

Since $RW_v^m(H) \subseteq CW_v^m(H)$ (by (10)), Claim 1 implies the following:

$$AW_v^m(H) = \{\} \ \vee \ |CW_v^m(H)| \geq 2. \tag{91}$$

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 (85)–(89). In Step 2, we show that H' also satisfies (82)–(84).

Step 1. We consider two cases.

First, if p is *not* deployed in H, then by (90), we have $p \in RW_v^m(H)$. In this case, let $H' = erase_p(H)$. By applying Lemma 4, it follows that H' is a regular computation in C with induction number m_H, satisfying (85)–(89). (Condition (B) of Lemma 4 follows from $p \in RW_v^m(H)$ and (91).)

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 (4) and (90), we have $(j, u) = (m, v)$, and hence we have $r \in AW_v^m(H)$. Thus,

- $AW_v^m(H)$ is nonempty, (92)

and by Claim 1, $RW_v^m(H)$ is also nonempty. Fix a process $q \in RW_v^m(H)$, and let $H'' = exchange_{pq}(H)$. By applying Lemma 3, 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:

$$P(H'') = P(H); \tag{93}$$
$$\mathrm{Act}^j(H'') = \mathrm{Act}^j(H); \tag{94}$$
$$\mathrm{Cvr}^j(H'') = \mathrm{Cvr}^j(H); \tag{95}$$
$$AW_w^j(H'') = AW_w^j(H); \tag{96}$$
$$CW_w^j(H'') = CW_w^j(H); \tag{97}$$
$$RW_w^j(H'') = \begin{cases} (RW_w^j(H) \cup \{p\}) - \{q\}, & \text{if } j = m \text{ and } w = v \\ RW_w^j(H), & \text{otherwise.} \end{cases} \tag{98}$$

By (98), we have $p \in RW_v^m(H'')$. Also, by (91), (92), and (97), we have $|CW_v^m(H'')| \geq 2$. That is,

$$p \in RW_v^m(H'') \ \wedge \ |CW_v^m(H'')| \geq 2. \tag{99}$$

Let $H' = erase_p(H'')$. We now apply Lemma 4 with 'H' $\leftarrow H''$. (Condition (B) of Lemma 4 follows from (99).) 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:

$$P(H') = P(H'') - \{p\}; \tag{100}$$
$$\mathrm{Act}^j(H') = \mathrm{Act}^j(H'') - \{p\}; \tag{101}$$
$$\mathrm{Cvr}^j(H') = \mathrm{Cvr}^j(H'') - \{p\}; \tag{102}$$
$$AW_w^j(H') = AW_w^j(H'') - \{p\}; \tag{103}$$
$$CW_w^j(H') = CW_w^j(H'') - \{p\}. \tag{104}$$

By combining (93)–(97) with (100)–(104), it follows that H' satisfies (85)–(89).

Step 2. We now show that H' constructed above satisfies (82)–(84). We prove (82) by proving its logical equivalent: if $CW_w^j(H')$ is empty and $CW_w^j(H)$ is nonempty, then $AW_w^j(H')$ is empty. By (89), the

antecedent of this implication implies $CW_w^j(H) = \{p\}$. However, by (4) and (90), this implies $(j,w) = (m,v)$, and hence, by (91), it follows that $AW_w^j(H)$ is empty. Thus, by (88), $AW_w^j(H')$ is also empty. It follows that H' satisfies (82).

Since $AW_v^m(H)$ and $CW_v^m(H)$ are disjoint (by (27)), by (90), we have $p \notin AW_v^m(H)$, and hence, by (88), we have $AW_v^m(H') = AW_v^m(H)$. Thus, since both H and H' have induction number m_H, by the definition of 'req' (given in (12)), we have

$$\mathrm{req}(m, v; H') = \mathrm{req}(m, v; H).$$

Also, by (89) and (90), we have

$$|CW_v^m(H')| = |CW_v^m(H)| - 1.$$

Combining these two assertions with the definition of $\pi(m,v)$ (given in (13)), (83) easily follows. (Note that $p \in CW_v^m(H)$ implies $(j,w) = (m,v)$.)

In order to prove (84), 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 (84) follows easily.

On the other hand, if (j,w) *is* a covering pair in H', then by (4), (90), and $(j,w) \neq (m,v)$, we have $p \notin CW_w^j(H)$, and hence, by (89), we have

$$CW_w^j(H') = CW_w^j(H).$$

Also, by (88), 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 (12)), we have

$$\mathrm{req}(j, w; H') \leq \mathrm{req}(j, w; H).$$

Combining these two assertions with (13), assertion (84) easily follows. □

We now formally present the chain erasing procedure, described in Section 4. Here, we denote the set of processes to erase by K. The procedure is shown in Figure 19. The following lemma proves its correctness.

Lemma 7 *Consider a regular computation H in C with induction number m_H, and a set K of processes. Assume the following:*

- $m_H \leq c - 2$, (105)
- $\pi_{\max}(H) = 0$, (106)
- $K \subseteq \mathrm{Cvr}(H)$, *and* (107)
- $|\mathrm{Act}(H)| \geq |K|/(c-1) + 2$. (108)

Then, there exists a regular computation H' in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H') < c$; (109)
- $|\mathrm{Act}(H')| \geq |\mathrm{Act}(H)| - |K|/(c-1)$; (110)
- *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*; (111)
  - *the following hold:*

$$P(H') \subseteq P(H) - K; \quad (112)$$
$$\mathrm{Act}^j(H') \subseteq \mathrm{Act}^j(H) - K; \quad (113)$$
$$\mathrm{Cvr}^j(H') \subseteq \mathrm{Cvr}^j(H) - K; \quad (114)$$
$$AW_w^j(H') \subseteq AW_w^j(H) - K; \quad (115)$$
$$CW_w^j(H') \subseteq CW_w^j(H) - K. \quad (116)$$

```
1:  F := H;      cnt := 0;
2:  for i := 1 to h do
        /* loop invariant:
            1. F is a regular computation in C with induction number m_H;
            2. π_max(F) < c
        */
3:      if p_i ∈ P(F) then
4:          erase p_i from F by applying Lemma 6 with 'H' ← F and 'p' ← p_i;
            let the resulting computation be F;
            cnt := cnt + 1;
5:          while π_max(F) = c do
                /* loop invariant: there exists exactly one covering pair (m, v) satisfying π(m, v; F) = c */
6:              choose a process r from AW^m_v(F);
7:              erase r from F by applying Lemma 6 with 'H' ← F and 'p' ← r;
                let the resulting computation be F
    od fi od
8:  H' := F
```

Figure 19: 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 $K \in \mathrm{Cvr}(H)$ holds. The correctness of this algorithm is formally proved in Lemma 7.

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 19. 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. (J1)
invariant For each segment index j and variable w, if $AW^j_w(F)$ is nonempty and $CW^j_w(F)$ is empty, then $CW^j_w(H)$ is also empty. (J2)
invariant $\pi_{\max}(F) \le c$. (J3)
invariant $(c-1) \cdot \big(|\mathrm{Act}(H)| - |\mathrm{Act}(F)|\big) + \pi(F) \le cnt$. (J4)

(Note that (J3) is weaker than the loop invariant $\pi_{\max}(F) < c$ stated in Figure 19, because (J3) holds throughout lines 2–8.) It is easy to see that line 1 establishes these invariants. In particular, (J3) and (J4) follow from (106). (Note that $\pi_{\max}(H) = 0$ implies $\pi(H) = 0$ by definition.) We now show that, for each line s ($2 \le s \le 7$) of the algorithm, if line s is executed while invariants (J1)–(J4) hold, then (J1)–(J4) also hold after the execution of line s. This will establish each of (J1)–(J4) as an invariant. Since F and cnt may be updated only by execution of lines 4 and 7, it suffices to check these two lines to prove the correctness of (J1)–(J4).

First, we claim that we can apply Lemma 6 at these lines. It suffices to show that the assumptions (78)–(81) stated in Lemma 6 are satisfied before executing either line 4 or 7. Assumptions (78) and (79) follow from (105) and (J3), respectively; (80) is guaranteed by lines 3 and 6. In order to obtain (81), note that $cnt \le h$ $(= |K|)$ is trivially an invariant, and that $\pi(F)$ is always nonnegative by definition. Thus, by (108) and (J4), we have $|\mathrm{Act}(F)| \ge |\mathrm{Act}(H)| - |K|/(c-1) \ge 2$, which implies (81).[17]

We now claim that execution of these lines preserves invariants (J1)–(J3). 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 6 implies that (J1) is preserved. (That is, if F_{old} satisfies (J1), then so does F_{new}.)

Also, by applying (82) and (88) with 'H' $\leftarrow F_{\mathrm{old}}$ and 'H'' $\leftarrow F_{\mathrm{new}}$, we have the following for each segment index j and variable w.

[17] The sole purpose of assumptions (81) and (108) is to prevent the degenerate case in which we happen to erase *all* active processes. (Clearly, our induction cannot proceed if we do not have any active processes.) Fortunately, as described in Section 4.3, such a case actually never occurs in our proof.

- If $AW^j_w(F_{\text{new}})$ is nonempty and $CW^j_w(F_{\text{new}})$ is empty, then $CW^j_w(F_{\text{old}})$ is also empty. (117)
- $AW^j_w(F_{\text{new}}) \subseteq AW^j_w(F_{\text{old}})$ holds. In particular, if $AW^j_w(F_{\text{new}})$ is nonempty, then so is $AW^j_w(F_{\text{old}})$. (118)

By (117) and (118), it follows that lines 4 and 7 preserve (J2). In particular, if $AW^j_w(F_{\text{new}})$ is nonempty and $CW^j_w(F_{\text{new}})$ is empty, then by (117) and (118), it follows that $AW^j_w(F_{\text{old}})$ is nonempty and $CW^j_w(F_{\text{old}})$ is empty. Since F_{old} satisfies (J2), this in turn implies that $CW^j_w(H)$ is empty.

In order to show that (J3) 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 $p \in CW^j_w(F_{\text{old}}) \ \wedge\ \pi(j, w; F_{\text{new}}) = \pi(j, w; F_{\text{old}}) + 1$ holds. (119)
- Line 4 is executed only if $\pi_{\max}(F_{\text{old}}) < c$ holds. (120)
- 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 $\pi(m, v; F_{\text{new}}) = c$. (121)
- 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:
  - $\pi(m, v; F_{\text{new}}) = 0$, (122)
  - there exists at most one covering pair (m', v') in F_{new} that satisfies $\pi(m', v'; F_{\text{new}}) = c$, and (123)
  - $\pi_{\max}(F_{\text{new}}) \le c$. (124)

Proof of (119)–(124)**:** Assertion (119) easily follows from applying (83) and (84) 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 (120) follows easily by inspecting the algorithm. In particular, line 1 establishes $\pi_{\max}(F) = 0$ by (106), and the **while** loop of lines 5–7 establishes $\pi_{\max}(F) < c$ upon termination.

We now prove (121). By the definition of $\pi_{\max}$ (given in (14)), $\pi_{\max}(F_{\text{new}}) = c$ implies that $\pi(m, v; F_{\text{new}}) = c$ holds for some covering pair (m, v). By (119) and (120), $\pi(m, v; F_{\text{new}}) = c$ may be established for at most one pair (m, v). We thus have (121).

We now prove (122). Consider the execution of line 7. By applying (88) 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 (12)), we have

$$\text{req}(m, v; F_{\text{new}}) = \text{req}(m, v; F_{\text{old}}) - c. \tag{125}$$

Moreover, since $AW^m_v(H)$ and $CW^m_v(H)$ are disjoint (by (27)), $r \in AW^m_v(F_{\text{old}})$ implies $r \notin CW^m_v(F_{\text{old}})$, and hence, by applying (89) as above, we have

$$CW^m_v(F_{\text{new}}) = CW^m_v(F_{\text{old}}). \tag{126}$$

Combining (125) and (126) with the definition of π (given in (13)), and using $\pi(m, v; F_{\text{old}}) = c$ (from the antecedent of (122)–(124)), we have $\pi(m, v; F_{\text{new}}) = 0$, and hence (122) follows.

Finally, the antecedent of (122)–(124) implies that $\pi(j, w; F_{\text{old}}) < c$ holds for each pair $(j, w) \ne (m, v)$. Hence, (123) and (124) easily follow from (119), (122), and the definition of $\pi_{\max}$ (given in (14)). □

We now prove that (J3) is an invariant. First, consider line 4. Combining (119) with the definition of $\pi_{\max}$, it follows that line 4 may increase $\pi_{\max}(F)$ by at most one. Therefore, by (120), line 4 establishes $\pi_{\max}(F_{\text{new}}) \le \pi_{\max}(F_{\text{old}}) + 1 \le c$, and hence preserves (J3).

Next, consider line 7. By combining (121), (123), and (124), it is easy to see that the loop invariant of the **while** loop (shown before line 6 in Figure 19) is indeed a correct invariant:

- Before the execution of lines 6 and 7, $\pi_{\max}(F_{\text{old}}) = c$ holds, and there exists exactly one covering pair (m, v) of F_{old} satisfying $\pi(m, v; F_{\text{old}}) = c$. (127)

Since this loop invariant implies the antecedent of (124), it follows that line 7 establishes $\pi_{\max}(F_{\text{new}}) \le c$, thereby maintaining (J3).

In order to show that (J4) is an invariant, we first need the following two properties.

- The execution of line 4 increases $\pi(F)$ by at most one. (128)
- The execution of line 7 decreases $\pi(F)$ by at least $c - 1$. (129)

Proof of (128) **and** (129)**:** Consider the execution of line s, where s is 4 or 7. By (119), one of the following holds.

(i) There exists exactly one pair (j, w) (where $1 \le j \le m_H$ and $w \in V$) satisfying $\pi(j, w; F_{\text{new}}) = \pi(j, w; F_{\text{old}}) + 1$. Moreover, for all other pairs $(l, u) \ne (j, w)$ (where $1 \le l \le m_H$ and $u \in V$), we have $\pi(l, u; F_{\text{new}}) \le \pi(l, u; F_{\text{old}})$.

(ii) For all pairs (l, u) (where $1 \le l \le m_H$ and $u \in V$), we have $\pi(l, u; F_{\text{new}}) \le \pi(l, u; F_{\text{old}})$.

By (i) and (ii), (128) easily follows. As for (129), note that, by (122) and (127), there exists a covering pair (m, v) of F_{old} satisfying $\pi(m, v; F_{\text{old}}) = c$ and $\pi(m, v; F_{\text{new}}) = 0$. Moreover, by (i) and (ii), the sum of ranks over all pairs *except for* (m, v) is increased by at most one by the execution of line 7. From these two observations, we have (129). □

Since line 4 erases a process p_i in K, by (3) and (107), we have $p_i \notin \text{Act}(H)$. Moreover, by applying (86) with 'j' $\leftarrow m_H$, 'H' $\leftarrow F_{\text{old}}$, '$H'$' $\leftarrow F_{\text{new}}$ each time Lemma 6 is used (at either line 4 or line 7), it easily follows that $\text{Act}(F)$ is always a subset of $\text{Act}(H)$. Hence, line 4 cannot change $\text{Act}(F)$. Since line 4 increases *cnt*, by (128), line 4 preserves (J4).

By applying (86) as above again, line 7 may decrease $|\text{Act}(F)|$ by at most one. Hence, by (129), line 7 preserves (J4). It follows that (J4) is indeed an invariant.

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 (85), each application of Lemma 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 (J1) and (J3) imply that, after the algorithm terminates, we obtain a regular computation H' in C with induction number m_H, satisfying (109). In particular, note that the **while** loop of lines 5–7 establishes $\pi_{\max}(F) \ne c$ upon termination. By (J3), it follows that line 8 establishes $\pi_{\max}(H') < c$, and hence we have (109).

Since the final value of *cnt* is at most $|K|$, and since $\pi(H')$ is nonnegative by definition, (J4) implies (110). Assertion (111) follows from (J2).

Finally, 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 (85)–(89), we have (112)–(116). □

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 4, in order to extend a regular computation, we must eliminate "conflicts" that are caused by appending these critical events. To facilitate this, we define a "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 (130) 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. (130)

Condition (C1): (erasing strategy)

- $Z^{\text{Cvr}} = \{\}$;
- for each $p \in Z^{\text{Act}}$, e_p is an atomic write event, writing a distinct variable;
- for each $p \in Z^{\text{Act}}$, $e_p \neq CS_p$;
- for each $p \in Z^{\text{Act}}$, if e_p writes a variable v, and if a process $q \neq p$ reads v in H, then $q \notin Z^{\text{Act}}$.

Condition (C2): (covering strategy)

- For each variable v, if $Z^{\text{Act}} \cap Z_v(H)$ is nonempty, then $|Z^{\text{Cvr}} \cap Z_v(H)| \geq c \cdot |Z^{\text{Act}} \cap Z_v(H)| + c^2$;
- for each $p \in Z^{\text{Act}} \cup Z^{\text{Cvr}}$, e_p is an atomic write event;
- for each $p \in Z^{\text{Act}} \cup Z^{\text{Cvr}}$, $e_p \neq CS_p$.

Condition (C3): (readers only)

- $Z^{\text{Cvr}} = \{\}$;
- for each $p \in Z^{\text{Act}}$, 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 $\text{Act}(H)$ into two disjoint sets $Z^{\text{Act}}(H)$ and $Z^{\text{Cvr}}(H)$ (one of which may possibly be empty), such that $\text{Act}(H) = Z^{\text{Act}}(H) \cup Z^{\text{Cvr}}(H)$ and the pair $(Z^{\text{Act}}(H), Z^{\text{Cvr}}(H))$ is conflict-free in H. □

In Lemma 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 $Z^{\text{Act}}(H)$ become active processes in the extended computation G, and processes in $Z^{\text{Cvr}}(H)$ become new covering processes to cover variables written by processes in $Z^{\text{Act}}(H)$. (Formally, we establish $\text{Act}(G) = Z^{\text{Act}}(H)$ and $\text{Cvr}(G) = \text{Cvr}(H) \cup Z^{\text{Cvr}}(H)$.)

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 8 *Let H be a regular computation in C with induction number m_H. Let $n = |\text{Act}(H)|$. Assume the following:*

- $m_H \leq c - 2$, (131)
- $\pi_{\max}(H) = 0$, *and* (132)
- $n \geq 7$. (133)

Then, there exists a regular computation F in C with induction number m_H, satisfying the following:

- $\pi_{\max}(F) \leq c$, (134)

- F *is conflict-free, and* (135)
- $|Z^{\text{Act}}(F)| \geq \dfrac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)}$. (136)

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' $\leftarrow H \mid p$, '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 12). That is, we choose a set Y such that

$$Y \subseteq \text{Act}(H); \tag{137}$$

$$e_p \neq CS_p, \quad \text{for all } p \in Y; \tag{138}$$

$$|Y| \geq (n-1)/2; \tag{139}$$

$$\big(\forall p : p \in Y :: e_p \text{ is a read event}\big) \quad \vee \quad \big(\forall p : p \in Y :: e_p \text{ is an atomic write event}\big). \tag{140}$$

We now erase processes in $\text{Act}(H)-Y$. (Note that Y is nonempty by (133) and (139).) Since $\text{Act}(H)-Y \subseteq \text{Act}(H)$, by applying Lemma 5 with 'K' $\leftarrow \text{Act}(H) - Y$, we can construct a regular computation H' in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H') \leq \pi_{\max}(H)$; (141)
- for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q,

$$\begin{aligned}
P(H') &= P(H) - (\text{Act}(H) - Y); && (142)\\
\text{Act}^j(H') &= \text{Act}^j(H) - (\text{Act}(H) - Y); && (143)\\
\text{Cvr}^j(H') &= \text{Cvr}^j(H); && (144)\\
AW_w^j(H') &= AW_w^j(H) - (\text{Act}(H) - Y); && (145)\\
CW_w^j(H') &= CW_w^j(H); && (146)\\
RW_w^j(H') &\supseteq RW_w^j(H).
\end{aligned}$$

By (132) and (141), and since $\pi_{\max}(H')$ is nonnegative by definition, we also have

$$\pi_{\max}(H') = 0. \tag{147}$$

By (137), and by applying (143) with 'j' $\leftarrow m_H$, we also have

$$\text{Act}(H') = \text{Act}^{m_H}(H') = \text{Act}^{m_H}(H) - (\text{Act}(H) - Y) = \text{Act}(H) - (\text{Act}(H) - Y) = Y. \tag{148}$$

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 $v_{\text{ce}}(p)$ to be the variable accessed by p's next critical event e_p. That is,

- for each $p \in Y$, e_p accesses $v_{\text{ce}}(p)$. (149)

For each $p \in Y$, we define $K(p) = q$ if q $(\neq p)$ is the single writer of $v_{\text{ce}}(p)$ in H' (see Figure 9). If H' has no single writer of $v_{\text{ce}}(p)$, or if p itself is the single writer of $v_{\text{ce}}(p)$, then we define $K(p) = \perp$. By definition,

$$K(p) \neq p, \quad \text{for all } p \in Y. \tag{150}$$

We now eliminate conflicts between Y and $\text{Cvr}(H')$ by applying the chain erasing procedure. Define CE, the "covering processes to be erased," as

$$CE = \{K(p) \colon p \in Y \ \wedge \ K(p) \in \text{Cvr}(H')\}.$$

Clearly, we have

$$|CE| \le |\{K(p)\colon p \in Y\}| \le |Y|, \quad \text{and} \tag{151}$$

$$CE \subseteq \mathrm{Cvr}(H'). \tag{152}$$

By (133), (139), and (148), we also have $|\mathrm{Act}(H')| = |Y| \ge 3 \ge 2(c-1)/(c-2)$ for large enough c. By (151), this in turn implies

$$|Y| \ge \frac{|Y|}{(c-1)} + 2 \ge \frac{|CE|}{(c-1)} + 2. \tag{153}$$

We now apply Lemma 7 with 'H' $\leftarrow H'$ and 'K' $\leftarrow CE$. Assumptions (105)–(108) stated in Lemma 7 follow from (131), (147), (152), and (148) and (153), respectively. We thus obtain a regular computation H'' in C with induction number m_H, satisfying the following:

- $\pi_{\max}(H'') < c$; (154)
- $|\mathrm{Act}(H'')| \ge |\mathrm{Act}(H') - CE| - |CE|/(c-1)$; (155)
- for each segment index j $(1 \le j \le m_H)$ 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; (156)
  - the following hold:

$$\begin{aligned} P(H'') &\subseteq P(H') - CE; && (157)\\ \mathrm{Act}^j(H'') &\subseteq \mathrm{Act}^j(H') - CE; && (158)\\ \mathrm{Cvr}^j(H'') &\subseteq \mathrm{Cvr}^j(H') - CE; && (159)\\ AW_w^j(H'') &\subseteq AW_w^j(H') - CE; && (160)\\ CW_w^j(H'') &\subseteq CW_w^j(H') - CE. && (161)\end{aligned}$$

Since $CE \subseteq \mathrm{Cvr}(H')$ holds by definition, CE and $\mathrm{Act}(H')$ are disjoint by (3), and hence, by (155), we also have

$$|\mathrm{Act}(H'')| \ge |\mathrm{Act}(H')| - |CE|/(c-1). \tag{162}$$

Define

$$Y' = \mathrm{Act}(H''). \tag{163}$$

By (148), and by applying (158) with 'j' $\leftarrow m_H$, we have

$$Y' = \mathrm{Act}^{m_H}(H'') \subseteq \mathrm{Act}^{m_H}(H') - CE = Y - CE. \tag{164}$$

Also, by applying (162), (148), (151), and (139) (in that order),

$$|Y'| \;\ge\; |\mathrm{Act}(H')| - \frac{|CE|}{c-1} \;=\; |Y| - \frac{|CE|}{c-1} \;\ge\; |Y| - \frac{|Y|}{c-1} \;=\; \frac{c-2}{c-1}|Y| \;\ge\; \frac{(c-2)(n-1)}{2(c-1)}. \tag{165}$$

We now construct an undirected graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y' (see Figure 13). For each process p in Y', we introduce edge $\{p, K(p)\}$ if $K(p) \in Y'$ holds. Since $K(p) \ne p$ (by (150)), 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 2, there exists an independent set $Z \subseteq Y'$ such that

$$|Z| \ge \frac{|Y'|}{3} \ge \frac{(c-2)(n-1)}{6(c-1)}, \tag{166}$$

where the latter inequality follows from (165). By (164), we also have

$$Z \subseteq Y' \subseteq Y - CE. \tag{167}$$

Since $Y' = \mathrm{Act}(H'') \subseteq P(H'')$ (by (3) and (163)), we have $Z \subseteq P(H'')$. Thus, by (157), we have

$$Z \subseteq P(H'') \subseteq P(H'). \tag{168}$$

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 (3), q is either in $\mathrm{Cvr}(H'')$ or $\mathrm{Act}(H'')$. If $q \in \mathrm{Cvr}(H'')$, then by applying (159) 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 (163) and (167), 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. □

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:

$$\begin{aligned} &Z = \bigcup_{v \in V} Z_v, \quad \text{and} \\ &Z_v \cap Z_u \neq \{\} \;\Rightarrow\; v = u. \end{aligned} \tag{169}$$

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

$$\begin{aligned} V_{\mathrm{HC}} &= \{v \in V \colon |Z_v| \geq 4c^2\}, \quad \text{and} \\ V_{\mathrm{LC}} &= \{v \in V \colon 0 < |Z_v| < 4c^2\}. \end{aligned}$$

Then, we have

$$|V_{\mathrm{HC}}| \leq \frac{|Z|}{4c^2}. \tag{170}$$

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:

$$P_{\mathrm{HC}} = \bigcup_{v \in V_{\mathrm{HC}}} Z_v, \quad P_{\mathrm{LC}} = \bigcup_{v \in V_{\mathrm{LC}}} Z_v. \tag{171}$$

Since the sets Z_v partition Z, P_{HC} and P_{LC} also partition Z:

$$Z = P_{\mathrm{HC}} \cup P_{\mathrm{LC}} \quad \wedge \quad P_{\mathrm{HC}} \cap P_{\mathrm{LC}} = \{\}. \tag{172}$$

We now construct subsets Z^{Act} and Z^{Cvr} of Z, such that the pair $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ 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 $Z^{\mathrm{Act}}(F) = Z^{\mathrm{Act}}$ and $Z^{\mathrm{Cvr}}(F) = Z^{\mathrm{Cvr}}$.)

Claim 2: There exist two disjoint subsets Z^{Act} and Z^{Cvr} of Z, such that $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ is conflict-free in H'', satisfying the following inequality:

$$|Z^{\mathrm{Act}}| \geq \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)}. \tag{173}$$

Proof of Claim: By Claim 1, any choice of $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ from Z satisfies (130). Thus, it suffices to show one of (C1)–(C3). By (140) and (167), 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 (166), 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 (173).

Case 2 (erasing strategy): Condition (W) is true, and $|P_{\text{HC}}| < |Z|/2$ holds.
By (166) and (172), we have

$$|P_{\text{LC}}| = |Z| - |P_{\text{HC}}| > \frac{|Z|}{2} \geq \frac{(c-2)(n-1)}{12(c-1)}. \tag{174}$$

Note that, by (171), 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 (174),

$$|V_{\text{LC}}| > \frac{|P_{\text{LC}}|}{4c^2} > \frac{(c-2)(n-1)}{48c^2(c-1)}. \tag{175}$$

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 (167), (172), (175), and the definition of Z_v, we have the following:

- $X \subseteq P_{\text{LC}} \subseteq Y'$ $(= \text{Act}(H''))$, (176)
- $|X| = |V_{\text{LC}}| > \dfrac{(c-2)(n-1)}{48c^2(c-1)}$, and (177)
- $v_{\text{ce}}(p)$ is distinct for each process $p \in X$. (178)

As explained in Section 4, 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 14). For each pair $\{p, q\}$ of different processes in X, we introduce edge $\{p, q\}$ if p reads $v_{\text{ce}}(q)$ in H''. By (176), each $p \in X$ executes m_H critical events in H'', and hence reads at most m_H distinct variables in H''. Therefore, by (178), 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 2, there exists an independent set $X' \subseteq X$ such that

$$|X'| \geq \frac{|X|}{2m_H+1} > \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)}, \tag{179}$$

where the latter inequality follows from (177). Also, by (178),

- $v_{\text{ce}}(p)$ is distinct for each process $p \in X'$, (180)

and by (149) 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 $q \notin X'$. (181)

Also, since by (138), (164), and (176), we have the following:

$$e_p \neq CS_p, \quad \text{for each } p \in X'. \tag{182}$$

We now define $Z^{\mathrm{Act}} = X'$ and $Z^{\mathrm{Cvr}} = \{\}$. By (179), we have (173). By (180)–(182) 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 (166), we have

$$|P_{\mathrm{HC}}| \geq \frac{|Z|}{2} \geq \frac{(c-2)(n-1)}{12(c-1)}. \tag{183}$$

By (171), P_{HC} is partitioned into subsets Z_v, for each $v \in V_{\mathrm{HC}}$. Moreover, by the definition of V_{HC},

$$|Z_v| \geq 4c^2, \quad \text{for each } v \in V_{\mathrm{HC}}. \tag{184}$$

Thus, we can partition each such Z_v into two disjoint sets Z_v^{Act} and Z_v^{Cvr}, such that the following holds:

$$Z_v = Z_v^{\mathrm{Act}} \cup Z_v^{\mathrm{Cvr}} \quad \wedge \quad Z_v^{\mathrm{Act}} \cap Z_v^{\mathrm{Cvr}} = \{\}; \tag{185}$$

$$|Z_v^{\mathrm{Act}}| = \left\lfloor \frac{|Z_v|}{c^2} \right\rfloor - 1. \tag{186}$$

By (169) and (184)–(186), we have the following:

$$Z_v^{\mathrm{Act}} \cap Z_u^{\mathrm{Act}} \neq \{\} \;\Rightarrow\; v = u, \quad \text{for each variable } v \text{ and } u; \tag{187}$$

$$|Z_v^{\mathrm{Act}}| > \frac{|Z_v|}{c^2} - 2 \geq \frac{|Z_v|}{c^2} - \frac{|Z_v|}{2c^2} = \frac{|Z_v|}{2c^2}; \tag{188}$$

$$|Z_v^{\mathrm{Cvr}}| = |Z_v| - |Z_v^{\mathrm{Act}}| \geq c^2 \cdot (|Z_v^{\mathrm{Act}}| + 1) - |Z_v^{\mathrm{Act}}| = (c^2 - 1) \cdot |Z_v^{\mathrm{Act}}| + c^2. \tag{189}$$

We can now define Z^{Act} and Z^{Cvr} as follows:

$$Z^{\mathrm{Act}} = \bigcup_{v \in V_{\mathrm{HC}}} Z_v^{\mathrm{Act}} \quad \text{and} \quad Z^{\mathrm{Cvr}} = \bigcup_{v \in V_{\mathrm{HC}}} Z_v^{\mathrm{Cvr}}. \tag{190}$$

By definition, the sets Z^{Act} and Z^{Cvr} partition P_{HC}:

$$P_{\mathrm{HC}} = Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}} \quad \wedge \quad Z^{\mathrm{Act}} \cap Z^{\mathrm{Cvr}} = \{\}; \tag{191}$$

Also, we have the following:

$$\begin{aligned}
|Z^{\mathrm{Act}}| &= \left|\textstyle\bigcup_{v \in V_{\mathrm{HC}}} Z_v^{\mathrm{Act}}\right| = \textstyle\sum_{v \in V_{\mathrm{HC}}} |Z_v^{\mathrm{Act}}| && \{\text{by (187)}\} \\
&> \sum_{v \in V_{\mathrm{HC}}} \frac{|Z_v|}{2c^2} && \{\text{by (188)}\} \\
&= \frac{1}{2c^2} \left|\textstyle\bigcup_{v \in V_{\mathrm{HC}}} Z_v\right| && \{\text{since each } Z_v \text{ is disjoint with each other, by (169)}\} \\
&= \frac{1}{2c^2} |P_{\mathrm{HC}}| && \{\text{by (171)}\} \\
&\geq \frac{(c-2)(n-1)}{24c^2(c-1)} && \{\text{by (183)}\} \\
&> \frac{(c-2)(n-1)}{48c^2(c-1)(2m_H+1)},
\end{aligned}$$

and hence we have (173). Finally, by (169) and (185), the sets Z_v are mutually disjoint, each partitioned into Z_v^{Act} and Z_v^{Cvr}. Thus, by (190), 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 (189), we have

$$|Z^{\mathrm{Cvr}} \cap Z_v| \geq (c^2 - 1) \cdot |Z^{\mathrm{Act}} \cap Z_v| + c^2 > c \cdot |Z^{\mathrm{Act}} \cap Z_v| + c^2, \tag{192}$$

where the last inequality follows from $c = \Theta(\log N)$ (given in (11)), assuming $N = \omega(1)$.

Also, since $P_{\mathrm{HC}} \subseteq Y$ (by (167) and (172)), by (138), we have the following:

$$e_p \neq CS_p, \quad \text{for each } p \in Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}. \tag{193}$$

Combining (192) and (193) with Condition (W), we have Condition (C2), and hence Claim 2 follows. □

Define $Z' = Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}$. By (167) and Claim 2, we have

$$Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}} = Z' \subseteq Z \subseteq Y' \subseteq Y - CE. \tag{194}$$

We now erase processes in $Y' - Z'$ by applying Lemma 5 with 'H' $\leftarrow H''$ and 'K' $\leftarrow Y' - Z'$. (Since Y' is defined to be $\mathrm{Act}(H'')$ in (163), and since, by (133) and (173), Z^{Act} is nonempty, we have $Y' - Z' \subsetneq \mathrm{Act}(H'')$. As noted prior to (154), H'' is a regular computation with induction number m_H.) We thus construct a regular computation F in C with induction number m_H, satisfying assertions (195)–(200), given below:

- $\pi_{\max}(F) \leq \pi_{\max}(H'')$; (195)
- for each j $(1 \leq j \leq m_H)$, k $(1 \leq k \leq m_H)$, variable w, and process q,

$$\begin{aligned} P(F) &= P(H'') - (Y' - Z'); & (196)\\ \mathrm{Act}^j(F) &= \mathrm{Act}^j(H'') - (Y' - Z'); & (197)\\ \mathrm{Cvr}^j(F) &= \mathrm{Cvr}^j(H''); & (198)\\ AW^j_w(F) &= AW^j_w(H'') - (Y' - Z'); & (199)\\ CW^j_w(F) &= CW^j_w(H''). & (200) \end{aligned}$$

By (163), (194), and by applying (197) with 'j' $\leftarrow m_H$, we also have

$$\mathrm{Act}(F) = \mathrm{Act}^{m_H}(F) = \mathrm{Act}^{m_H}(H'') - (Y' - Z') = Y' - (Y' - Z') = Z'. \tag{201}$$

We now claim that F satisfies (134)–(136). By (154) and (195), we have (134). Since $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ is conflict-free in H'' (by Claim 2), it is also conflict-free in F. Define $Z^{\mathrm{Act}}(F) = Z^{\mathrm{Act}}$ and $Z^{\mathrm{Cvr}}(F) = Z^{\mathrm{Cvr}}$. Since $Z' = Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}}$ by definition, by (201), $\mathrm{Act}(F)$ is partitioned into two disjoint sets $Z^{\mathrm{Act}}(F)$ and $Z^{\mathrm{Cvr}}(F)$, such that $(Z^{\mathrm{Act}}(F), Z^{\mathrm{Cvr}}(F))$ is conflict-free in F. Thus, F is conflict-free by definition, so we have (135). Finally, by (173), we have (136). □

The following lemma extends a conflict-free computation, thus providing the induction step that leads to the lower bound in Theorem 3.

Lemma 9 *Let H be a regular computation in C with induction number m_H. Assume the following:*

- $m_H \leq c - 2$, (202)
- *H is conflict-free, and* (203)
- $\pi_{\max}(H) \leq c$. (204)

Since H is conflict-free, by definition, $\mathrm{Act}(H)$ *is partitioned into two disjoint sets $Z^{\mathrm{Act}} = Z^{\mathrm{Act}}(H)$ and $Z^{\mathrm{Cvr}} = Z^{\mathrm{Cvr}}(H)$.*

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* (205)
- $\mathrm{Act}(G) = Z^{\mathrm{Act}}$. (206)

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:

$$Z^{\text{Act}} \cup Z^{\text{Cvr}} = \text{Act}(H) \quad \text{and} \quad Z^{\text{Act}} \cap Z^{\text{Cvr}} = \{\}. \tag{207}$$

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 $v_{\text{ce}}(p)$. (208)

Define Z_v, the set of active processes that access v via their next critical events, as $Z_v = \{p \in Z\colon v_{\text{ce}}(p) = v\}$. Arbitrarily index processes in Z^{Act} as

$$Z^{\text{Act}} = \{p_1,\ p_2,\ \ldots,\ p_b\}, \tag{209}$$

where $b = |Z^{\text{Act}}|$. In order to construct the new $(m_H + 1)^{\text{st}}$ segment E, for each $p \in Z^{\text{Act}}$, 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 $p \in Z^{\text{Act}}$ (see Figure 6).

Step 1. For each m $(1 \le m \le m_H)$, if $ce(p, m)$ 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 q from RW_v^m, and deploy q by adding $ie(q, m)$, its invocation event on v, to $C(p)$. (After deployment, q becomes a deployed process, $dp(p, m)$, and does not belong to RW_v^m any more.)

Step 2. If Condition (C2) (from the definition of a conflict-free pair, given before Lemma 8) is true, then let $v = v_{\text{ce}}(p)$. We choose a process q from Z^{Cvr} such that its next critical event e_q is a write to v. (Thus, we have $p \in Z^{\text{Act}} \cap Z_v$ and $q \in Z^{\text{Cvr}} \cap Z_v$.) We then deploy q by adding $ie(q, m_H + 1)$, its invocation event on v, to $C(p)$. (Similarly, q becomes a deployed process, $dp(p, m_H + 1)$.)

We claim that there exist enough reserve processes to deploy throughout the construction of all next covering segments. By Lemma 1, for each covering pair (m, v) of H, we have $|RW_v^m(H)| > |AW_v^m(H)|$. (Assumptions (16) and (17) stated in Lemma 1 follow from (202) and (204), respectively.) Thus, H has enough reserve processes to use in Step 1. Also, if Condition (C2) is true, then we have $|Z^{\text{Cvr}} \cap Z_v| \ge c \cdot |Z^{\text{Act}} \cap Z_v| + c^2 > |Z^{\text{Act}} \cap Z_v|$, and hence we have enough processes in $Z^{\text{Cvr}} \cap Z_v$ to use in Step 2.

We now construct E as follows:

$$E = S(p_1, m_H + 1) \circ C(p_1) \circ S(p_2, m_H + 1) \circ C(p_2) \circ \cdots \circ S(p_b, m_H + 1) \circ C(p_b). \tag{210}$$

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 2.)

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:

$$\mathit{last_writer}(v, F_1) = p \ \vee\ \mathit{last_writer}(v, F_1) = \bot \ \vee\ \mathit{value}(v, F_1) = \star.$$

Proof of Claim: If f_p is an event in H, then by applying Lemma 2 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 (210)),

- f_p is contained in $S(p, m_H + 1)$, (211)

and also $p \in Z^{\text{Act}}$ holds. By (207), we also have

$$p \in Z^{\text{Act}} \subseteq \text{Act}(H). \tag{212}$$

Since f_p reads v, p executes a critical read of v in G. Thus,

- $ce(p, m)$ reads v, for some m $(1 \leq m \leq m_H + 1)$. (213)

Let $g_q = \mathit{last_writer_event}(v, F_1)$. If we have either $q = p$ or $q = \perp$, then we are done. Thus, assume $q \neq p \,\wedge\, q \neq \perp$. 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 (31) and (210),

- g_q is contained in solo segment $S(q, l)$, for some l $(1 \leq l \leq m_H + 1)$. (214)

Thus,

- $ce(q, j)$ is a write to v, for some $j \leq l$; (215)

We consider two cases, depending on the value of j.

Case 1: $j = m_H + 1$.

In this case, by (215),

- $e_q = ce(q, m_H + 1)$ is a write to v. (216)

Moreover, by (214) and (215), g_q is contained in $S(q, m_H + 1)$, and hence, by (209) and (210), we have

$$q \in Z^{\mathrm{Act}}. \tag{217}$$

Thus, by (203), (216), and the definition of a conflict-free computation,

- either Condition (C1) or Condition (C2) is true,

and hence, by (212), $e_p = ce(p, m_H + 1)$ is also a write event. Therefore, by (213), we have $m \leq m_H$, and hence,

- p reads v in H. (218)

By (212), (216), (217), and (218), we have a contradiction of (the last line of) Condition (C1). Thus, assume that Condition (C2) is true. By (208) and (216), it follows that Step 2 (in the construction of E) adds an invocation event h on v to $C(q)$. Therefore, by (211), 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 $g_q = \mathit{last_writer_event}(v, F_1)$.

Case 2: $1 \leq j \leq m_H$.

In this case, by (215), 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. (219)

If $1 \le m \le m_H$, then by (213), and by applying R4 with 'p' $\leftarrow q$, 'q' $\leftarrow p$, and 'm' $\leftarrow j$, we have $p \in \mathrm{Cvr}^j(H)$. However, since $\mathrm{Cvr}^j(H) \subseteq \mathrm{Cvr}(H)$ (by (6)), and since $\mathrm{Act}(H)$ and $\mathrm{Cvr}(H)$ are disjoint (by (3)), this contradicts (212).

Therefore, by (213), $m = m_H + 1$ holds, and $e_p = ce(p, m_H + 1)$ reads v. Thus, by (130) and (219), we have $q \in \mathrm{Act}(H) - (Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}})$. However, this is impossible by (207).

Second, assume that q is not the single writer of H, *i.e.*, $CW_v^j(H)$ is nonempty. If $l \le m_H$, then by (214), 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 (209), (210), and (214), 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 (211), 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 $g_q = \mathit{last_writer_event}(v, F_1)$. □

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 ($1 \le m \le m_H + 1$) and variable v:

$$\mathrm{Act}^m(G) = \begin{cases} \mathrm{Act}^m(H), & \text{if } m \le m_H \\ Z^{\mathrm{Act}}, & \text{if } m = m_H + 1; \end{cases} \tag{220}$$

$$\mathrm{Cvr}^m(G) = \begin{cases} \mathrm{Cvr}^m(H), & \text{if } m \le m_H \\ \mathrm{Cvr}(H) \cup Z^{\mathrm{Cvr}}, & \text{if } m = m_H + 1; \end{cases} \tag{221}$$

$$AW_v^m(G) = \begin{cases} AW_v^m(H), & \text{if } m \le m_H \\ Z^{\mathrm{Act}} \cap Z_v, & \text{if } m = m_H + 1 \text{ and Condition (C1) or (C2) is true} \\ \{\}, & \text{if } m = m_H + 1 \text{ and Condition (C3) is true;} \end{cases} \tag{222}$$

$$CW_v^m(G) = \begin{cases} CW_v^m(H), & \text{if } m \le m_H \\ Z^{\mathrm{Cvr}} \cap Z_v, & \text{if } m = m_H + 1 \text{ and Condition (C2) is true} \\ \{\}, & \text{if } m = m_H + 1 \text{ and Condition (C1) or (C3) is true.} \end{cases} \tag{223}$$

From (210) and the construction of the next covering segments, assertions (2)–(10) and (220)–(223) follow immediately. The construction of the next covering segments ensures that G satisfies R1 and R2. From (203) 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 (203), $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ satisfies (130) with respect to H. By (220), we have $q \in Z^{\mathrm{Act}}$. Thus, applying (130) with 'p' $\leftarrow q$ and 'q' $\leftarrow p$, and using $p \neq q$ 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 (207).

Second, assume that $m = m_H + 1$. Since $p \in \mathrm{Act}^{m_H+1}(G)$, by (220), 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 ($1 \leq j \leq m_H + 1$) and a process $q \in \mathrm{Act}^j(G)$ different from p. Our proof obligation is to show the following three conditions:

(i) if $j \leq m_H$ and $ce(q, j)$ is a write to v, then $CW_v^j(G)$ is nonempty;

(ii) if $j \leq m_H$ 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, $(Z^{\mathrm{Act}}, Z^{\mathrm{Cvr}})$ satisfies (130) with respect to H, and hence we have either $p = q$ or $q \in \mathrm{Act}(H) - (Z^{\mathrm{Act}} \cup Z^{\mathrm{Cvr}})$. The former contradicts our assumption, and the latter is impossible by (207).

Second, consider (ii). In this case, the last line of Condition (C1) implies $q \notin Z^{\mathrm{Act}}$, which in turn implies $q \in \mathrm{Cvr}^{m_H+1}(G)$ by (207), (220), and (221).

Finally, consider (iii). By (220), $q \in \mathrm{Act}^{m_H+1}(G)$ implies $q \in Z^{\mathrm{Act}}$. Combining this with $p \in Z^{\mathrm{Act}}$, and using the second line of Condition (C1), (iii) follows easily.

Finally, we claim that G satisfies (205) and (206). From (220), we have (206). (Note that Z^{Act} is defined to be $Z^{\mathrm{Act}}(H)$.) In order to show (205), we must show $\pi(m, v; G) = 0$ for every covering pair (m, v) in G. We consider two cases.

First, if $m \leq m_H$, then by (204), we have

$$\pi(m, v; H) \leq c.$$

Since $m \leq m_H$, we have $AW_v^m(G) = AW_v^m(H)$ and $CW_v^m(G) = CW_v^m(H)$. Hence, by (12), and since G has an induction number of $m_H + 1$, we also have

$$\mathrm{req}(m, v; G) = \mathrm{req}(m, v; H) - c.$$

Combining these two assertions with the definition of 'π' (given in (13)), 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 (222) and (223), we have $AW_v^m(G) = Z^{\mathrm{Act}} \cap Z_v$ and $CW_v^m(G) = Z^{\mathrm{Cvr}} \cap Z_v$. Hence, by (C2), we have $|CW_v^m(G)| \geq c \cdot |AW_v^m(G)| + c^2 > c \cdot (|AW_v^m(G)| + c - m_H) = \mathrm{req}(m, v; G)$, and hence $\pi(m, v; G) = 0$ follows. □

Theorem 3 *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 $H_1 = S(1,1) \circ S(2,1) \circ \cdots \circ S(N,1) = \langle Enter_1,\ Enter_2,\ \ldots,\ Enter_N \rangle$, where $P = \{1, 2, \ldots, N\}$. By the definition of a mutual exclusion system, $H_1 \in C$. In H_1, each process p becomes a "single writer" of its auxiliary variable $entry_p$. By checking conditions (2)–(10) and R1–R4 individually, it follows that H_1 is a regular computation with induction number 1.

We repeatedly apply Lemma 8 and Lemma 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 (131) or (133) of Lemma 8 is not satisfied.

Define $n_m = |\mathrm{Act}(H_m)|$ for each m. Applying Lemma 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 (136).) Applying Lemma 9 with 'H' $\leftarrow F_m$, we construct H_{m+1} such that $\mathrm{Act}(H_{m+1}) = Z^{\mathrm{Act}}(F_m)$ (by (206)). Combining these relations, we have

$$n_{m+1} \geq \frac{(c-2)(n_m-1)}{48c^2(c-1)(2m+1)},$$

and hence, by (11), (131), and (133),

$$n_{m+1} \geq \frac{a' n_m}{m \log^2 N} \geq \frac{a n_m}{\log^3 N},$$

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

$$\log n_k \geq \log N - 3(k-1)\log\log N + (k-1)\log a. \tag{224}$$

If we stop the induction at step k because assumption (131) is not satisfied, then we have $k = c-2$, and hence, by (11), $k = \Theta(\log N)$ holds. On the other hand, if we stop the induction because assumption (133) is not satisfied, then we have $n_k < 7$, and hence, by (224),

$$\log 7 > \log N - 3(k-1)\log\log N + (k-1)\log a,$$

which in turn implies

$$k > \frac{\log N - \log 7}{3\log\log N - \log a} + 1 = \Theta\left(\frac{\log N}{\log\log N}\right).$$

Therefore, in either case, we have $k = \Omega(\log N/\log\log N)$. Since H_{k-1} satisfies (133), we can choose a process p from $\mathrm{Act}(H_{k-1})$ that executes exactly $k-1$ solo segments (and hence, $k-1$ critical events) in H_{k-1}. Thus, $H_{k-1} \,|\, p$ is a solo computation that satisfies the theorem. □