2-26-2010

Constant RMR Solutions to Reader Writer Synchronization

Vibhor Bhatt
Dartmouth College

Prasad Jayanti
Dartmouth College

Dartmouth Computer Science Technical Report TR2010-662
Constant RMR Solutions to Reader Writer Synchronization*

Vibhor Bhatt and Prasad Jayanti
Department of Computer Science
Dartmouth College
vibhor, prasad@cs.dartmouth.edu

February 26, 2010

Abstract

We study *Reader-Writer Exclusion* [1], a well-known variant of the Mutual Exclusion problem [2] where processes are divided into two classes–*readers* and *writers*–and multiple readers can be in the Critical Section (CS) at the same time, although no process may be in the CS at the same time as a writer. Since readers don't conflict with each other, they should not obstruct each other. Specifically, the *concurrent entering* property must be satisfied: if all writers are in the remainder section, each reader should be able to enter the CS in a bounded number of its own steps. Three versions of the Reader-Writer Exclusion problem are commonly studied–one where writers have priority over readers, another where readers have priority, and the last where neither class has priority over the other and no process may starve.

To ensure high performance on Cache-Coherent (CC) and Distributed Shared Memory (DSM) multiprocessors, algorithms should be designed to generate as few remote memory references (RMRs) as possible. The ideal would be to achieve constant RMR complexity, i.e., the worst case number of RMRs that a process generates in order to enter and exit the CS once is a constant, independent of the number of processes.

Constant RMR complexity algorithms have existed for Mutual Exclusion for two decades [3, 4], but none exists for Reader-Writer Exclusion. Danek and Hadzilacos' lower bound proof implies that it is impossible to achieve sublinear RMR complexity for DSM machines [5]. For CC machines, the best existing bound, also due to Danek and Hadzilacos [5], is $O(\log n)$, where n is the number of processes. In this work, we present the first constant RMR complexity algorithms for all three versions of the Reader-Writer Exclusion problem (for CC machines).

*Submitted to the Proceedings of the Twenty-Ninth annual symposium on Principles of distributed computing (PODC 2010).

1 Introduction

Mutual exclusion [2] is a well-studied, fundamental problem in distributed computing. Here processes repeatedly cycle through four sections of code–Remainder Section, Try Section, Critical Section (CS), and Exit Section–in that order, and the problem consists of designing the code for the Try and Exit sections so that the mutual exclusion property–at most one process is in the CS at any time–is satisfied. The algorithm is often required to additionally satisfy some liveness (e.g., starvation-freedom) and fairness (e.g., first-come-first-served [6]) properties.

Past research on mutual exclusion can be cleanly divided into two parts. The early research, spanning roughly 1965-89, focused mostly on formulating desirable properties (livelock freedom versus starvation freedom, first-come-first served, abortability, self-stabilization *etc*.) and on designing algorithms that realized these properties on the machines of that era–the multitasking/timesharing uniprocessors. (For a survey of this early research, see [7].) These algorithms, however, are poorly suited to multiprocessors because they do not take into account the fact that if a processor p accesses a shared variable that resides at p's local memory module, the access will be fast, but if p accesses a remote shared variable, the access can be extremely slow (because of the delay in gaining exclusive access to the interconnection bus and the high latency of the bus). Later research (1990-present) on locking algorithms has therefore been driven by the goal to minimize the number of *remote memory references (RMRs)* (see the survey [8]). (In *Distributed Shared Memory* (DSM) machines, a reference to a shared variable X is considered remote if X is at a memory module of a different processor; in *Cache Coherent* (CC) machines, a reference to X by a process p is considered remote if X is not in p's cache.) This goal implies that locking algorithms should be designed to spin locally, i.e., processes do not make any remote references in busywait loops. The ideal goal is to design locking algorithms whose *RMR complexity*–the worst case number of remote memory references made by a process to enter and exit the CS once–is a constant, independent of the numbers of readers/writers attempting to acquire the lock. Research in this direction over the last two decades led to many algorithms [3, 4, 9, 10, 11, 12, 13, 14, 15, 16, 17], lower bounds [18, 19, 20, 21], and an understanding of the limitations of various shared memory primitives [18, 19]. The highlight is the design of constant RMR complexity mutual exclusion algorithms: Anderson's algorithm achieves constant RMR complexity for CC machines [3] and Mellor-Crummey and Scott's algorithm, which was awarded the Dijkstra Prize in 2006, achieves constant RMR complexity for both CC and DSM machines [4].

In this paper, we study *Readers/Writers Exclusion* [1], a well-known variant of the Mutual Exclusion problem where processes are divided into two classes–*readers* and *writers*–and the exclusion property is revised to allow for more concurrency: multiple readers can be in the CS at the same time, although no process may be in the CS at the same time as a writer. Since readers don't conflict with each other, it is desirable if they don't obstruct each other from entering the CS. More specifically, if all writers are in the remainder section, every reader in the Try section should be able to enter the CS in a bounded number of its own steps. This property is known in the literature as *concurrent entering* [22, 23]. Reader-writer locks are used extensively in operating systems and in parallel applications to implement shared data structures, where processes whose operations modify the state are modeled as writers and processes that merely sense the state as readers.

As already mentioned, constant RMR complexity is achievable for Mutual Exclusion. Can a similar result be achieved for Reader/Writer Exclusion? For DSM machines, the answer is no: Danek and Hadzilacos' lower bound proof for 2-Session Group Mutual Exclusion implies that a sublinear RMR complexity algorithm satisfying concurrent entering is impossible for the Reader/Writer Exclusion problem, even if there is only one writer [5]. Intuitively, if the writer w is in the CS and all n readers are spinning on shared variables in their respective local memory modules, when w leaves the CS it must wake each reader r by writing in r's spin location, thereby incurring $\Omega(n)$ RMR complexity. This argument does not apply to CC machines since all readers can potentially spin on the same location and the writer can wake all of them at once by writing into that single location. Henceforth, we pursue the design of constant RMR complexity reader/writer algorithms only for CC machines.

Existing reader/writer algorithms either fail to satisfy concurrent entering [1, 9, 24, 25] or have non-constant RMR complexity [1, 5, 26]. Using a novel synchronization scheme that works very differently from the earlier ones, we design the first constant RMR complexity algorithm for Reader/Writer Exclusion. Besides concurrent entering, our algorithm satisfies many desirable properties, including starvation-freedom, FCFS among writers, FIFE (First-In First-enabled) among readers, and bounded exit.

Starting from the earliest paper [1], most works on Reader/Writer Exclusion studied two more natural variants of the problem–one in which readers have priority over writers, and the other where writers have priority. None of the works, however, gave any specification (formal or informal) of what it means for one class of processes to have priority over the other. As we explain later, it is nontrivial to provide a rigorous specification that captures our intuitive notion of priority. We give such a specification for both the reader-priority and the writer-priority cases, and design constant RMR algorithms for the two cases. To the best of our knowledge, the algorithms presented in this paper are the first to achieve constant RMR complexity for the three cases of reader-priority, writer-priority, and no-priority.

2 Specification of the Reader-Writer Problem

In the *Reader-Writer Problem*, each process is labeled a *reader* or a *writer*, and its program is an infinite loop that cycles through four sections of code—Remainder section, Try section, Critical Section (CS), and Exit section. The Try section, in turn, consists of two code fragments—a *doorway*, followed by a *waiting room*—with the requirement that the doorway is a bounded "straight line" code [6]. Intuitively, a process "registers" its request for the CS by executing the doorway, and then busywaits in the waiting room until it has the "permission" to enter the CS. Initially, all processes are in their remainder section. We assume that no process accesses shared variables while in the CS or the remainder section.

Each execution of the Try, CS, and Exit sections by a process is called an *attempt*; it is a read attempt (respectively, write attempt) if the process is a reader (respectively, writer). An attempt by a process p spans from a time t when p executes the first state statement of its Try section to the earliest time $t' > t$ when p completes the Exit section and goes back to the remainder section. An attempt A is *active* in a configuration C if, in C, A is in the Try section, CS, or Exit section.

The following definitions of "precedence" and "enabled" will be useful for defining fairness and priority properties in the next section. If A is an attempt by a process p, henceforth we write "A completes the doorway at time t" as a shorthand for "p completes the doorway at time t during the attempt A."

Definition 1 If A and A' are any two attempts in a run (possibly by different processes), A *doorway precedes* A' if A completes the doorway before A' begins executing the doorway. A and A' are *doorway concurrent* if neither doorway precedes the other.

Definition 2 A process p is *enabled to enter the CS* in configuration C if p is in the Try section in C and there is an integer b such that, in all runs from C, p enters the CS in at most b of its own steps.

The *Reader-Writer Problem* is to design the code for the Try and Exit sections so that certain safety and liveness properties hold in all runs of the algorithm. The exact set of properties will depend on whether we want readers to have a higher priority than writers, or writers to have a higher priority than readers, or neither. We begin by stating the properties that are desirable in all three cases.

- (P1). Mutual Exclusion : If a writer is in the CS at any time, then no other process is in the CS at that time.

- (P2). Bounded Exit : There is an integer b such that in every run, every process completes the Exit section in at most b of its steps.

- (P3). First-Come-First-Served (FCFS) among writers : If w and w' are any two write attempts in a run and w doorway precedes w', then w' does not enter the CS before w.

- (P4). First-In-First-Enabled (FIFE) among readers : Let r and r' be any two read attempts in a run such that r doorway precedes r'. If r' enters the CS before r, then r is enabled to enter the CS at the time r' enters the CS.

- (P5). Concurrent Entering : Informally, if all writers are in the remainder section, readers should be able to enter the CS in a bounded number of steps. More precisely: There is an integer b such that, if σ is any run from a reachable configuration such that all writers are in the remainder section in every configuration in σ, then every read attempt in σ executes at most b steps of the Try section before entering the CS.

Finally, we state the liveness property. Starvation-freedom is too strong a property to require in our context: when one class of processes (e.g., readers) have priority over the other class (e.g., writers), the starvation of processes belonging to the lower priority class is unavoidable. Therefore, we require the weaker *livelock freedom* property, which is appropriate in all three cases of reader-priority, writer-priority, and no-priority. It guarantees that, under the standard assumption that no process occupies the CS forever and no process permanently stops taking steps in the Try or the Exit section, not all processes will get stuck in the Try or the Exit section.

Henceforth, we say a process p *crashes* in an infinite run σ if p enters the CS and does not subsequently leave the CS or p has an incomplete attempt and only a finite number of steps in σ.

- (P6). Livelock Freedom : If no process crashes in an infinite run, then infinitely many attempts complete in that run.

2.1 Specifying reader priority

Suppose that we want to give readers priority over writers. Then, if a reader r completes the doorway before a writer w even enters the doorway, it makes sense to require that w does not enter the CS before r. What else must we ensure? Consider a scenario where a writer p is in the CS, and a reader r and a writer w are waiting. If readers have priority over writers, when p leaves the CS, we expect r, and not w, to enter the CS, even if w has been waiting much longer than r. If p were instead a reader, we still expect that r enters the CS before w. We capture these expectations through the *reader-priority relation* $>_{rp}$, which is a binary relation between the set of read attempts and the set of write attempts in a run.

Definition 3 Let r and w be a read attempt and a write attempt, respectively, in a run. We define $r >_{rp} w$ if

- r doorway precedes w, or

- There is a time when some reader or writer is in the CS, r is in the waiting room, and w is in the Try section.

The next property states what it means for readers to have priority over writers.

- (RP1). Reader Priority : Let r and w be a read attempt and a write attempt, respectively, in a run. If $r >_{rp} w$, then w does not enter the CS before r.

2.2 Specifying writer priority

Specifying writer priority is similar, but not entirely symmetric. To see why, consider a scenario where a process p is in the CS, and a reader r and a writer w are waiting. If p is a writer, when p leaves the CS, we expect w, and not r, to enter the CS. However, if p is a reader, it might not be possible to stop r from proceeding to the CS (because r might be already enabled to enter the CS by the FIFE property). So, requiring w to enter the CS before r would contradict FIFE. In view of these considerations, we define the *writer-priority relation* $>_{wp}$ as follows.

Definition 4 Let r and w be a read attempt and a write attempt, respectively, in a run. We define $w >_{wp} r$ if

- w doorway precedes r, or
- There is a time when some writer is in the CS, w is in the waiting room, and r is in the Try section.

The next property formalizes what it means for writers to have priority over readers.

- (WP1). Writer Priority : Let r and w be a read attempt and a write attempt, respectively, in a run. If $w >_{wp} r$, then r does not enter the CS before w.

2.3 The case of no priorities

If readers or writers have priority over the other, a process belonging to the lower priority class may starve. However, if neither class has priority over the other, we can require that no process—neither a reader nor a writer—starves:

- (P7). Starvation Freedom : If no process crashes in an infinite run, then all attempts complete in that run.

2.4 Some additional desirable properties

2.4.1 When readers have priority

Consider the case when readers have priority over writers. The Reader Priority property stated above ensures that a writer does not race ahead of a higher priority reader, but it gives no guarantee about the ability of the higher priority reader to make progress towards entering the CS. As a result, certain undesirable scenarios can arise. For example, consider a scenario where a reader r' is in the CS, a reader r and a writer w are in the waiting room, and all other processes are in the remainder section. By definition, $r >_{rp} w$; hence, by Reader Priority property, w does not enter the CS before r. Now, suppose that r' chooses to remain in the CS while r and w repeatedly execute steps in the waiting room. Since the CS is already occupied by a reader and r is actively taking steps, we would want r to be able to enter the CS and co-occupy the CS with r'. Yet, the properties listed above do not guarantee that r will be able to join r' in the CS (more specifically, r's entry into the CS is not guaranteed until r' leaves the CS). To fix this weakness in the specification, we now define an additional property. The first part of the property states that, if the CS is occupied by readers at a time when some reader r is in the waiting room, then r must be able to enter the CS in a bounded number of its own steps. The second part states that if there is no writer in the CS or the Exit section, a reader r is in the waiting room, and r has higher priority than every writer in the Try section, then r must be able to enter the CS in a bounded number of its own steps.

- (RP2). Unstoppable Reader Property : Let C be any reachable configuration in which some read attempt r is in the waiting room.

1. If a reader is in the CS in C, then r is enabled to enter the CS in C.
2. If no writer is in the CS or the Exit section in C and $r >_{rp} w$ holds for all write attempts w that are in the Try section in C, then r is enabled to enter the CS in C.

Note that neither part implies the other. Since the first part demands that r be enabled even if some writers are present in the Exit section, it does not follow from the second part. Since the first part requires the CS to be occupied by one or more readers and the second part does not, the second part does not follow from the first.

2.4.2 When writers have priority

Arguing as above, we will now identify a property for the case when writers have priority over readers. Since there is a fundamental difference between readers and writers—unlike readers, writers cannot share the CS—the property below will be quite different than the one for the reader-priority case.

To motivate the property, suppose that the CS and Exit sections are empty, a writer w is in the waiting room (all other writers are in the remainder section), and $w >_{wp} r$ for all readers r in the Try section. Then, we would want the writer w to be able to enter the CS in a bounded number of its own steps.

Next, consider a slightly different scenario in which two writers w and w' are both in the waiting room (all other writers are in the remainder section), we have both $w >_{wp} r$ and $w' >_{wp} r$ for all readers r in the Try section, and, as before, the CS and the Exit sections are empty. Since all active readers are dominated by writers (by the $>_{wp}$ relation), we would want to stipulate that readers cannot block w or w' from entering the CS. But can one of these writers block the other? The answer will depend on whether one writer clearly dominates the other: if one writer doorway precedes the other, then the former should be able to enter the CS in a bounded number of its own steps. On the other hand, if neither w nor w' doorway precedes the other, neither process may be enabled to enter the CS; however, in this case, if both w and w' keep taking steps, it is reasonable to require that one of the two will eventually enter the CS, regardless of whether other writers and readers are slow, fast, or have crashed. These considerations lead to:

- (WP2). Unstoppable Writers Property : Let C be any reachable configuration in which (1) no processes are in the CS or the Exit section, (2) the set S of write attempts in the waiting room is non-empty, (3) $w \in S$ is the earliest among the write attempts in S to enter the waiting room, and (4) $S' \subseteq S$ is the set of write attempts in S that are doorway concurrent with w. If $w >_{wp} r$ for all active read attempts r in C, then in every infinite run σ from C, either some write attempt in S' enters the CS or some write attempt in S' has only finitely many steps. (In other words, if *all* of the write attempts in S' keep taking steps, then one of them is guaranteed to eventually enter the CS, regardless of whether other processes perform their steps fast, slow, or even crash.)

Note that this property, together with Mutual Exclusion, implies that the first process to enter the CS in any run from C will be from S'. To see why, suppose that some process $p \notin S'$ enters the CS first in some run from C. Let C' be a configuration in which p is in the CS, and let σ' be the finite run from C to C'. Consider an infinite run σ'' from C' in which all processes in S' repeatedly execute steps and all other processes (including p) take no steps. By the above property, some write attempt in S' enters the CS in σ'', contradicting Mutual Exclusion (since p is already in the CS).

3 Single Writer Algorithm satisfying Starvation Freedom and Writer Priority

In this section, we present an algorithm that supports only a single writer and multiple readers, and satisfies starvation freedom as well as writer priority properties. The algorithm in the next section is also a single writer

algorithm that satisfies reader priority properties. In section 5, we show how to transform these single-writer algorithms into multi-writer algorithms.

Figure 1 presents a single writer algorithm that satisfies properties (P1)-(P7), (WP1) and (WP2). Hence, it addresses the case when the writer has priority over readers while also ensuring that no process–the writer or a reader–starves. The overall idea is as follows. The writer can enter the critical section from two sides, 0 and 1. It never changes its side during one attempt of the critical section. The writer also toggles its side for every new attempt. To enter from a certain side, say 1, the writer sets the shared variable *D* to 1. Then it waits for the readers from the previous side (in this case side 0) to exit the critical and exit section. The last reader to exit from side 0, lets the writer in the critical section. Once the writer is done with the critical section of side 1, it lets the readers waiting from side 1 into the critical section, using variable *Gate*$[1]$ described later.

The readers, in their try section set their side d equal to *D*. Then they increment their count in side d and attempt for critical section from side d. In order to enter the critical section from side d, they busy wait on *Gate*$[d]$ until it is open. When the readers are exiting they decrement their count from side d and the last exiting reader wakes up the writer. Now we describe the shared variables used in the algorithm.

Shared Variables

D $\in \{0, 1\}$ is a read/write variable, initialized to 0
ExitPermit is a boolean read/write variable
$\forall d \in \{0, 1\}$, *Permit*$[d]$ is a boolean read/write variable
$\forall d \in \{0, 1\}$, *Gate*$[d]$ is a boolean read/write variable, *Gate* [0] is initialized to *true* and *Gate* [1] to *false*
EC is a *F&A* variable with two components [*writer-waiting* $\in \{0, 1\}$, *reader-count* $\in \mathbb{N}$], initialized to $[0, 0]$
$\forall d \in \{0, 1\}$, $C[d]$ is a *F&A* variable with two components [*writer-waiting* $\in \{0, 1\}$, *reader-count* $\in \mathbb{N}$], initialized to $[0, 0]$

procedure `Write-lock()`

1. REMAINDER SECTION
2. *prevD* $\leftarrow$ *D*, *currD* $\leftarrow \overline{\textit{prevD}}$
3. *D* $\leftarrow$ *currD*
4. *Permit*[*prevD*] $\leftarrow$ *false*
5. if (*F&A*$(C[\textit{prevD}], [1, 0]) \neq [0, 0]$)
6. **wait till** *Permit*[*prevD*]
7. *F&A*$(C[\textit{prevD}], [-1, 0])$
8. *Gate*[*prevD*] $\leftarrow$ *false*
9. *ExitPermit* $\leftarrow$ *false*
10. if (*F&A*$(\textit{EC}, [1, 0]) \neq [0, 0]$)
11. **wait till** *ExitPermit*
12. *F&A*$(\textit{EC}, [-1, 0])$
13. CRITICAL SECTION
14. *Gate*[*D*] $\leftarrow$ *true*

procedure `Read-lock()`

15. REMAINDER SECTION
16. $d \leftarrow$ *D*
17. *F&A*$(C[d], [0, 1])$
18. $d' \leftarrow$ *D*
19. if($d \neq d'$)
20. *F&A*$(C[d'], [0, 1])$
21. $d \leftarrow$ *D*
22. if(*F&A*$(C[\overline{d}], [0, -1]) = [1, 1]$)
23. *Permit*$[\overline{d}] \leftarrow$ *true*
24. **wait till** *Gate*$[d]$
25. CRITICAL SECTION
26. *F&A*$(\textit{EC}, [0, 1])$
27. if(*F&A*$(C[d], [0, -1]) = [1, 1]$)
28. *Permit*$[d] \leftarrow$ *true*
29. if(*F&A*$(\textit{EC}, [0, -1]) = [1, 1]$)
30. *ExitPermit* $\leftarrow$ *true*

Figure 1: Single-Writer Multi-Reader Algorithm satisfying Starvation Freedom and Writer-Priority

3.1 Shared variables and their purpose

All the shared variable names start with upper case and the local variables start with lower case.

$\underline{D}$: This is a boolean read/write variable, and is only written by the writer. This variable denotes the side with which the writer wants to attempt for the critical section. It is toggled every time in the beginning of the try section by the writer. The readers use this variable to determine which side they should attempt for critical section.

$\underline{Gate[d]}$, $d \in \{0, 1\}$: This is a read/write boolean variable, written only by the writer. *Gate*$[d]$ is used by to block the readers to enter the critical section through side d. So before entering the critical section all the readers have to check if the *Gate*$[d]$ is open, where d is the side the reader is attempting for critical section. One invariant maintained by the algorithm regarding *Gate* is that before the writer starts with the attempt to enter the critical section from side d (sets *D* to d), the *Gate*$[d]$ is already closed, i.e., *Gate*$[d]$ = *false*. Also the writer once done with its critical section from side d opens the *Gate*$[d]$ for the readers who entered through side d and are waiting.

$\underline{Permit[d]}$, $d \in \{0, 1\}$: This is a read/write boolean variable written and read by both readers and the writer. The writer busy waits on *Permit*$[d]$ to get the permission to go to critical section from the readers who are present from side d. The idea is that the last reader to exit side d, will wake up the writer using *Permit*$[d]$.

$\underline{ExitPermit}$: This is a read/write boolean variable written and read by both readers and the writer. It is similar to *Permit*, with the difference that it it used by the writer to wait on the readers to clear up the exit section. The idea is that the last reader to exit the exit section will wake up the writer using this variable.

$\underline{C[d]}$, $d \in \{0, 1\}$: This is a fetch&add variable, read and updated both by the writer and readers. $C[d]$ has two components [*writer-waiting*, *reader-count*]. Where *writer-waiting* $\in \{0, 1\}$ denotes whether the writer is waiting for the readers from side d to leave the critical section, it is only updated by the writer. And *reader-count* denotes the number of readers currently registered in side d. *reader-count* of $C[d]$ is only incremented (and decremented) by a reader once in any attempt. The idea is that the last process to leave side d and get $[1, 1]$ from a *F&A*$(C[d], -1)$ will wake up the writer through *Permit*$[d]$.

$\underline{EC}$: This is a fetch&add variable, read and updated both by the writer and readers. Similar to $C[d]$, it has two components [*writer-waiting*, *reader-count*]. Where *writer-waiting* $\in \{0, 1\}$ denotes whether the writer is waiting for the readers to complete the exit section. And *reader-count* denotes the number of readers currently in the exit section. Similar to $C[d]$, the idea is that the last process to leave the exit section and get $[1, 1]$ from *F&A*$(EC, -1)$ will wake up the writer through *ExitPermit*.

3.2 Line by Line commentary

First we describe the code of the writer w in `Write-lock()` procedure. First in the bounded doorway, Line 2-3, the writer toggles *D*, *currD* is set to the new side and *prevD* is set to the previous side. This informs the new reader to attempt the critical section from side *D* = *currD*. As mentioned earlier *Gate*[*currD*] at this point is closed, so any reader which enters the try section now will be blocked till w completes the current attempt of critical section from side *currD*. Hence, if w doorway precedes a reader r, then r does not enter the critical section before w. This gives us the writer priority property (WP1).

In order to enter the critical section, w has to be sure that there are no readers are in critical section from the previous side. This is because the algorithm maintains the invariant that no readers are present in the critical section from the side *currD*, we'll discuss this fact again. Now w sets *Permit*[*prevD*] to *false* (Line 4). And then it increments the *writer-waiting* component of C[*prevD*] to set it to 1 (Line 5). This will notify the readers that w is waiting on *Permit*[*prevD*]. If w finds C[*prevD*] equal to $[0, 0]$ at Line 5, it knows that all the readers from side *prevD* have left, hence it does not't have to wait for them at Line 6.

If w goes to Line 6, it waits until it gets permission from the readers in side *prevD*, i.e., *Permit*[*prevD*] = *true*. Only the last reader from the side *prevD* will do this. Hence if w goes past Line 6, it knows for sure that there

is no reader in critical section. Now w decrements the *writer-waiting* component of $C[prevD]$ to set it to 0 (Line 7). Now all the active readers are either in side *currD* or are about to register for it. So one can see that next time the writer attempts to try for critical section with side *prevD* there will be no readers in critical section with side *prevD*. Then w closes *Gate*[*prevD*] for its next attempt to critical section (Line 8). In Lines 9 to 12, w waits for all the processes to leave the exit section in the same way it waited for them to leave the critical section in Lines 4-6. We will explain this subtle feature of the algorithm later. Note that at this point no reader is in the critical or exit section. Now the writer can to go critical section (Line 13). Finally, w exits by waking up the waiting readers by opening *Gate*[*currD*].

Now we proceed to describe the code of a reader r in `Read-lock()`. First r sets the value of d to *D* and then increments the *reader-count* of $C[d]$ (Line 16,17). Then it checks the value of *D* again (Line 18). If it is different, it proceeds with the *if* statement starting at Line 20 and increments on the other side, i.e., d'. And then it reads the value of *D* into d (Line 21). Now as r incremented the *reader-count* of both $C[d]$ and $C[\overline{d}]$, it decrements on the other side to which it does not belong, i.e., $\overline{d}$ (Line 22). But if the value returned to r after the *F&A*$(C[\overline{d}])$ is equal to $[1, 1]$, then it means the *reader-count* of $C[\overline{d}]$ is set to 0 and the writer is waiting on *Permit*$[\overline{d}]$. In this case r should wake up the writer who is waiting on *Permit*$[\overline{d}]$.

Now when r is at Line 24, it is only registered at side d. It waits until the *Gate*$[d]$ is open. If the writer is in the remainder section, then *Gate*$[d]$ will be open, hence concurrent entry of readers is satisfied. Once *Gate*$[d]$ opens, the reader enters the critical section (Line 25). In the exit section the reader first increments the exit-count, i.e., *EC* (Line 26). And then it decrements the $C[d]$ (Line 27). Similar to Line 22, depending upon the return value of the *F&A* operation at Line 27, r wakes up the writer at Line 28. Similarly it decrements *EC* at Line 29, and wakes up the writer if r is the last exiting reader and the writer is waiting on *ExitPermit* (Line 30).

3.3 Subtle features of the algorithm

In the above description of the algorithm we deferred the explanation of a subtle feature, namely, the need for the writer to wait for readers in the exit section. We now explain why this feature is need to guarantee mutual exclusion. Say the writer w does not wait for all the readers to finish the exit section before entering the critical section. Consider the following scenario. The writer w is at Line 6 waiting for *Permit*$[0]$ to set to be *true*, and there are only two reader r and r' active in the system. Say r is in critical section, and say r' is at Line 17 and it set $d = 0$ long time ago. By our discussion earlier as w is in try section with side 1, d of r is equal to 0. Now say, r exits the critical section and executes Line 27, hence, $\text{PC}_r = 28$ and $C[0] = [1, 0]$. Now say r' starts executing and one can see from the Line 17-21, that it will increment both $C[d]$ and $C[\overline{d}]$ and set its d equal to 1. So now $C[0] = [1, 1]$ and when r' executes Line 22, it will get $[1, 1]$ as the return value and then execute Line 23. And now the w wakes up, and if w does not wait for r to exit, r is poised to set *Permit*$[0]$ equal to *true* for a future writer. Now one can see that mutual exclusion could be in danger.

Theorem 1 (Single-Writer Multi-Reader lock with Starvation Freedom and Writer Priority) *The algorithm in Figure 1 implements a Single-Writer Multi-Reader lock satisfying the properties (P1)-(P7), (WP1) and (WP2). The RMR complexity of the algorithm in the CC model is O(1). The algorithm uses O(1) number of shared variables that support read, write, and fetch&add operations.*

4 Single Writer Algorithm satisfying Reader Priority

A single-writer multi-reader algorithm is presented in Figure 2 which satisfies properties (P1)-(P6), (RP1) and (RP2). The overall idea of the algorithm is as follows. For the writer w to enter the critical section it has to set a shared variable X to *true*. This variable X is not equal to *true* in the beginning of w's try section. To set X to *true*, w executes the procedure called `Promote`. The purpose `Promote` is to set X to *true* if no readers are

active (reader-count is zero). If some reader is active and w is not able to set X to *true*, w busy waits till all the readers exit and the last one of them wakes w up. Note that the readers can keep going to the critical section and starve the writer forever, but this is acceptable as this is a reader priority algorithm.

Readers in their try section first increment the count C. Then they check if X is *true*. If it is not, they know the writer is not in the critical section, so they just enter the critical section. If X is *true* then the writer is in the critical section, so they wait on a variable (*Gate*), till the writer wakes them up. When the readers exit, they decrement the count C and then execute the `Promote` procedure just as w executed it in its try section. Now we describe the shared variables used in the algorithm.

Shared Variables

$D \in \{0, 1\}$ is a read/write variable, initialized to 0
$\forall d \in \{0, 1\}$, *Gate*$[d]$ is a boolean read/write variable, *Gate* [0] is initialized to *true* and *Gate* [1] to *false*
$X \in \mathcal{PID} \cup \{true\}$ is a *CAS* variable , initialized to any $i \in \mathcal{PID}$
Permit is a boolean read/write variable, initialized to *true*
C is a *fetch and add* variable, initialized to 0

procedure `Write-lock`$_i()$

1. REMAINDER SECTION
2. $D \leftarrow \overline{D}$
3. *Permit* $\leftarrow$ *false*
4. `Promote()`
5. **wait till** *Permit*
6. CRITICAL SECTION
7. *Gate*$[\overline{D}] \leftarrow$ *false*
8. *Gate*$[D] \leftarrow$ *true*
9. $X \leftarrow i$

procedure `Promote()`

10. $x = X$
11. if$(x \neq$ *true*$)$
12. if$($*CAS*$(X, x, i))$
13. if$(\neg$*Permit*$)$
14. if$(C = 0)$
15. if$($*CAS*$(X, i,$ *true*$))$
16. *Permit* $\leftarrow$ *true*

procedure `Read-lock`$_i()$

17. REMAINDER SECTION
18. *F&A*$(C, 1)$
19. $d \leftarrow D$
20. $x \leftarrow X$
21. if $(x \in \mathcal{PID})$
22. *CAS*(X, x, i)
23. if $(X =$ *true*$)$
24. **wait till** *Gate*$[d]$
25. CRITICAL SECTION
26. *F&A*$(C, -1)$
27. `Promote()`

Figure 2: Single-Writer Multi-Reader Algorithm satisfying Reader Priority

4.1 Shared variables and their purpose

All the shared variable names start with upper case letter and the local variables with lower case letters.

$\underline{D}$: This is a boolean read/write variable, and is only written by the writer. This variable denotes the side with which the writer wants to attempt for the critical section. The writer toggles at the beginning of its try section.

$\underline{Gate[d]}$, $d \in \{0, 1\}$: This is a read/write boolean variable, written only by the writer. *Gate*$[d]$ is used to block the readers to enter the critical section through side d. Also once the writer is done with its critical section from side d, it opens the *Gate*$[d]$ for the readers in side d. The algorithm maintains the following two invariants with

respect to *Gate* variables : (invariant 1) if the writer is in the remainder section, then $Gate[D] = true$, (invariant 2) if the writer is in the try section and not the critical section then $Gate[\overline{D}] = true$. Another invariant maintained by the algorithm is that both $Gate[0], Gate[1]$ are not *true* at any time.

$\underline{X}$: This is a *CAS* variable which either stores *pid* of a process or *true*. $X \neq true$ means that the writer is not in critical section. When the writer begins its try section the value of X is not *true*. Before the writer enters the critical section, X is set to *true* either by the writer itself or the last reader to exit critical section. The algorithm maintains the following invariant with respect to X : (invariant 3) if a reader is in the critical section then either $X \neq true$ or $X = true$, $Gate[D] = true$ and the writer is at Line 9.

$\underline{C}$: This is a fetch&add variable, read by the writer and incremented only by readers. C is only incremented by a reader in the beginning of the try section and decremented in the exit section.

Permit : This is a read/write boolean variable written and read by both readers and the writer. The writer busy waits on *Permit* to get the permission from the readers to go to the critical section. The writer in the try section or the exiting reader checks if $C = 0$ in the `Promote` procedure. If it finds that $C = 0$, then it first attempts to set the variable X to *true*, and if it succeeds, it sets $Permit = true$ in order to enable the writer to enter the critical section.

4.2 Line by Line commentary

Now we informally describe the code for the writer w in `Write-lock`. First the writer toggles *D* and sets it to a new value, call it d (Line 2). At this point $Gate[d] = false, X \neq true$ and $Permit = true$. Then it sets *Permit* to *false* (Line 3). Then it executes the `Promote` procedure (Line 4). We will go into the detail of `Promote` later. Then w busy waits on *Permit* until it is set to *true* (Line 5). Once *Permit* is set to *true*, w enters the critical section (Line 6). In the exit section w closes *Gate* on the other side $\overline{D}$ for the next attempt by the writer for critical section (Line 7). In Line 8, w opens $Gate[d]$ for the readers waiting from side d. At this point the critical section belongs to the readers, and they can go freely in it even though $X = true$. As $Gate[D]$ is open the invariant (3) is not violated. And finally w sets X to its own *pid* before finishing the exit section (Line 9).

Now we describe the code for a reader r in `Read-lock`. First r increments the counter C (Line 18). Then sets its side d as the value in *D* (Line 19). What if the writer changes *D* after r reads it ? We claim that if *D* was changed by a writer w after r reads it, then r should still be fine, rather w cannot enter the critical section before r. This is true because at the time w changes *D*, the variable $X \neq true$. For w to enter the critical section, some process has to execute `Promote` and check $C = 0$ before setting X to *true*. But as r has already incremented C, this would not happen and hence w cannot enter the critical section before r. This also demonstrates that the algorithm satisfies Reader Priority (RP1). In the next three Lines r tries to set X to its own pid if X is not already *true* (Lines 20-22). We will explain the need of this subtle feature later. Then r checks the value of X (Line 23). If X is not *true* then r knows that the writer is not in critical section hence it enters the critical section in Line 25. If X is *true* at Line 23, then r busy waits on $Gate[d]$ till it is open (Line 24). Once that gate is open r enters the critical section (Line 25). In the exit section r first decrements the counter C and then executes `Promote`.

The purpose of the `Promote` is to enable the writer to enter the critical section if no readers are in the try or critical section. Now we explain the code of `Promote` as executed by a process p_i whose pid is i. First p_i sets its local variable x to X and checks if x is *true* (Line 10-11). If X is *true* then there is nothing for p_i to do as the process which set X to *true* will wake up the writer. If $x \neq true$, then p_i tries to *CAS* its own pid (i) into X (Line 12). Then p_i checks if $Permit = false$ (Line 13), if it is not then it knows that the writer is not waiting on *Permit*, so there is nothing for p_i to do. Then it checks if $C = 0$ (Line 14). If $C = 0$ then it tries to *CAS true* into X by comparing X with i (15). If the *CAS* succeeds then p_i wakes up the writer by setting *Permit* to *true*. One might wonder, why can't p_i just *CAS true* into X without *CAS*ing i into X. This is subtle feature of the algorithm and we will address this later.

4.3 Subtle features of the algorithm

Now we explain the need for the subtle features mention earlier, namely, (A) the need for Lines 20-22 and (B) the need to *CAS* in the pid in to X in `Promote`. We need both the features to ensure the mutual exclusion property.

To see why we need (A), consider the following scenario. No reader is active in the system and the writer w starts its try section and executes up to Line 15 in `Promote`. One can easily see that this would happen. Now a reader r starts its try section. If Lines 20-22 are not there, since as $X \neq$ *true*, r will enter the critical section. Now if w executes Line 15, its *CAS* would be successful and hence it will set *Permit* to *true* and enter the critical section also. Hence mutual exclusion would be violated. The same scenario can arise when a reader is about to execute Line 15, and then some reader quickly starts and completes its doorway. So having Lines 20-22 ensure that even if some reader r starts the try section while some process is about to execute Line 15, mutual exclusion is not violated.

Now we explain why we need feature (B). Assume that Line 12 is not there and Line 15 is *CAS*$(X, x,$ *true*$)$. Now, assume the writer is waiting for *Permit* at Line 5 and two reader p_i and p_j with pids i and j respectively are at Line 10. Assume that no other readers are active. Now assume that both p_i and p_j execute Lines 10-15 together and they find $x = r$, where r is the pid of some reader p_r. Now say p_i succeeds in doing the *CAS* at Line 15, and p_j is still at Line 15. Now say later, in future X is again set to r. If p_i executes Line 15 now, it will we poised to set *Permit* to *true* irrespective of the number of readers in critical section, thus putting mutual exclusion to danger.

The main theorem stated below summarizes the properties of the single-writer multi-reader lock implemented by the algorithm in Figure 2. Its formal proof is given in the appendix.

Theorem 2 (Single-Writer Multi-Reader lock with Reader Priority) *The algorithm in Figure 2 implements a Single-Writer Multi-Reader lock satisfying the properties (P1)-(P6), (RP1) and (RP2). The RMR complexity of the algorithm in the CC model is $O(1)$. The algorithm uses $O(1)$ number of shared variables that support read, write, fetch&add, and compare&swap operations.*

5 Single Writer to Multi Writer Constructions

In this section we use the single algorithms described before to the full multi-writer versions while preserving all the properties the single writer algorithms have. For our construction we will use a existing mutual exclusion lock, M. This lock will be used by the writers so that only one writer accesses the underlying single writer algorithm at any time. The lock we use was designed by T. Anderson [3]. Apart from satisfying mutual exclusion, starvation freedom, FCFS and bounded exit, it has $O(1)$ RMR in CC model. The lock also satisfies the following property which is used to guarantee Unstoppable Writer Priority (WP2) in the writer priority algorithm : If a set S of processes is in the waiting room and no process is in the critical or exit section then, some process in S is enabled for critical section.

Notations used in this section : We denote *SW-Write-try* (respectively, *SW-Read-try*) for the try section code of the writer (respectively, reader) in the corresponding single writer algorithm. For example when we say *SW-Write-try* from the algorithm in Figure 1, we mean the try section of the writer given in the Figure 1 (Lines 2-12). Similarly we use *SW-Write-exit* (respectively, *SW-Read-exit*) for the exit section code of the writer (respectively, reader). We use the procedures *acquire* and *release* for the try and exit section of M. A process returns from *acquire*(M) only after it owns the lock. It owns the lock M till it executes *release*(M).

We claim that the transformation $\mathcal{T}$ implements a multi-writer single-reader, starvation free lock satisfying (P1)-(P7) given any single-writer single-reader lock satisfying (P1)-(P7). The transformation also implements a multi-writer single-reader, reader priority lock satisfying (P1)-(P6), (RP1) and (RP2) given any single-writer

procedure `Write-lock()`

1. REMAINDER SECTION
2. `acquire`(M)
3. *SW-Write-try*()
4. CRITICAL SECTION
5. *SW-Write-exit*()
6. `release`(M)

procedure `Read-lock()`

7. REMAINDER SECTION
8. *SW-Read-try*()
9. CRITICAL SECTION
10. *SW-Read-exit*()

Figure 3: $\mathcal{T}$: transformation from a Single Writer Multi Reader to Multi Writer Multi Reader algorithm

single-reader lock satisfying (P1)-(P6), (RP1) and (RP2). The following theorems states these facts formally and we leave their proof as an exercise for the readers.

Theorem 3 (Multi-Writer Multi-Reader lock with Starvation Freedom) *The algorithm in Figure 3 implements a Multi-Writer Multi-Reader lock satisfying the properties (P1)-(P7) using the lock M from [3] and the algorithm in Figure 1. The RMR complexity of the algorithm in the CC model is $O(1)$. The algorithm uses $O(n)$ number of shared variables that support read, write, and fetch&add.*

Theorem 4 (Multi Writer Multi Reader lock with Reader Priority) *The algorithm in Figure 3 implements a Multi Writer Multi Reader lock satisfying the properties (P1)-(P6), (RP1) and (RP2) using the lock M from [3] and the algorithm in Figure 2. The RMR complexity of the algorithm in the CC model is $O(1)$. The algorithm uses $O(n)$ number of shared variables that support read/write, fetch&add and compare&swap operations.*

Wcount $\in [0 \ldots n]$ is *F&A* variable
W-token $\in \{\mathcal{PID} \cup \mathit{false} \cup \{0,1\}\}$ is *CAS* variable
D, *Gate* are the shared variables used in *SW-waiting-room*

procedure `Write-lock()`

1. REMAINDER SECTION
2. *F&A*(*Wcount*, 1)
3. $t =$ *W-token*
4. if $(t \in \mathcal{PID})$
5. *CAS*(*W-token*, t, *false*)
6. $t =$ *W-token*
7. if $(t \in \{0,1\})$
8. $D \leftarrow t$
9. `acquire`(M)
10. $\mathit{currD} \leftarrow D, \mathit{prevD} \leftarrow \overline{\mathit{currD}}$
11. if (*W-token* $\in \{0,1\}$)
12. **wait** till *Gate*[*prevD*]
13. *SW-waiting-room*()
14. CRITICAL SECTION
15. *W-token* $\leftarrow p$
16. *F&A*(*Wcount*, -1)
17. `release`(M)
18. if(*Wcount* $= 0$)
19. if(*CAS*(*W-token*, p, *prevD*))
20. *Gate*[*currD*] $=$ *true*

Figure 4: `Write-lock()` of the Multi-Writer Multi-Reader Writer Priority Algorithm, `Read-lock()` procedure is same as in Figure 3

5.1 Multi-Writer Multi-Reader Writer Priority

Say we use $\mathcal{T}$ together with the single-writer, writer priority algorithm given in Figure 1. The transformation $\mathcal{T}$ together with $SWWP$ does not work for the writer-priority case. To see why, take the following scenario :

Let w be a writer in critical section and w' be a writer in the waiting room. Now say a reader r enters the try section and then the waiting room. Now w starts exiting and executes *SW-Write-exit*, Line 5 of $\mathcal{T}$. At this point the the reader r is enabled and it would enter the critical section, violating the desired writer priority. In Figure 4 we give our correct multi-writer multi-reader, writer priority algorithm. We first describe the shared variables used in the algorithm and their roles.

The algorithm uses the single-writer multi-reader writer priority lock given in Figure 1. We will refer to this lock as $SWWP$. Also *SW-waiting-room* corresponds to the Lines $4-12$ of $SWWP$. Apart from the shared variables used in $SWWP$, following additional shared variables are used in the algorithm.

Wcount : This is a *F&A* variable and it keeps the count of the writers in the try and the critical section. The writers increment it in try section and decrement it in the exit section.

W-token : This is a *CAS* variable, which can store either a $0, 1,$ *false* and a process id. A writer in exit section first writes its own pid into *W-token* and then it checks if any writers are in try section. If there are writers in the try section then it leaves *W-token* as it is. If the exiting writer does not notice any writers in the try section, it tries to *CAS* in d into *W-token*, where d is the new side with which the writers should attempt the critical section in $SWWP$. The writer in try section tries to *CAS false* into *W-token* in try section if it sees that *W-token* contains some pid. We will get into its detail later.

5.2 Informal description of the algorithm in Figure 4

The overall idea is as follows. The readers simply executed the $SWWP$ protocol. For a writer w to be in critical section it needs to be in the critical section of both lock M and $SWWP$ and in that order. When w is exiting and notices some process in try section (using *Wcount*), it does not exit $SWWP$, but exits the lock M. This writer w leaves a token for the next writer w' so that w' does not participate in $SWWP$. The situation gets tricky if w does not see any writer in try section and is just about to exit $SWWP$ when a writer and bunch of readers enter the try section.

We handle this tricky situation in this algorithm as follows. If a writer w' in the try section knows that the last exiting writer w exited $SWWP$, w' executes the doorway of $SWWP$ before entering the waiting room of M. Before we go into the details of this, we make the following observation about $SWWP$: any reader r which starts its try section after a writer W has executed Line 3, i.e., $D \leftarrow$ *currD*, does not enter the critical section before W. So the writer w' executes the doorway of $SWWP$ by setting *D* to the new side with which the writers should attempt for critical section. w' gets this side from the *W-token* which was set by the exiting writer w.

The `Read-lock` procedure is same for the readers as in $SWWP$, so we describe the $\texttt{Write}-\texttt{lock}_p()$ as executed by a process p in detail. In order to describe the code of the writer w in $\texttt{Write}-\texttt{lock}()$ clearly we will do Line by Line commentary starting from its critical section. Say the writer w is in the critical section. Let *currD* be the current side of w in $SWWP$. In the exit section w first sets *W-token* to its own pid p (Line 15). Then w decrements *Wcount* (Line 16). At this point *Wcount* is the number of writers in the try section. Then w releases the lock M (Line 17). Now w has to check if any writers are in the try section. If there are no writers in in the try section, w has to exit $SWWP$ so as to avoid readers to starve forever in absence of any future writers. To achieve this w first checks if *Wcount* is equal to zero (Line 18). If *Wcount* $= 0$, then w tries to *CAS* in *W-token*, the next direction with which the writers should attempt for critical section (Line 19). Note that if w is a very old writer, i.e., many writers have have entered the critical section after w, the *CAS* of w would fail as *W-token* would be equal to some other pid. If the *CAS* succeeds, then w exits $SWWP$ (Line 20). Note that the exit section of the

writer has only one statement, *Gate*[*currD*] $\leftarrow$ *true*. If the *CAS* at Line 19 succeeds, how is w confident that no process is in the waiting room ? To understand this lets look at the try section of a writer w'.

First w' increments *Wcount* (Line 2). Then w' checks if *W-token* is equal to pid of some process (Line 3-4). If *W-token* is equal to some pid, then it might be the case that that some writer w is in exit section and is about to execute Line 19. To preempt w from giving the critical section to the readers, w' tries to *CAS false* into *W-token* (Line 5). Now this ensures that at the time *CAS* by w succeeds at Line 19, no process is in the waiting room.

But what if the writer w' enters the waiting room while w is at Line 20 ? To ensure writer priority, w' should not be overtaken (into the critical section) by a reader r which starts the try section after w' enters the doorway. So before entering the waiting room, w' checks if the *CAS* by the last exiting writer w succeeded at Line 19, i.e., *W-token* $\in \{0, 1\}$ (Line 6-7). If *W-token* $\in \{0, 1\}$, then w' executes the doorway of $SWWP$. In this case it is just setting *D* to the new direction which is given by *W-token* (Line 8). Now w' enters the waiting room of $acquire(M)$. As it has succeeded in completing the doorway of $SWWP$, no reader which starts its doorway now can go past w'. Then w' waits to acquire M (Line 9). Once w' owns M, it sets the local variables *currD* and *prevD* appropriately for the underlying single writer protocol, i.e., $SWWP$ (Line 10). If w' is the first writer to acquire lock M after the previous writer has exited $SWWP$, w' has to compete in $SWWP$. So w' checks if *W-token* $\in \{0, 1\}$ (Line 11). If *W-token* $\in \{0, 1\}$ then w' first waits for the previous writer to exit from underlying protocol (Line 12). We have to do this because it is possible for the previous writer w to succeed in its *CAS* at Line 19, but not exit from $SWWP$ (Line 20). So if w' does not wait for w to execute Line 20, w might cause danger to the safety property later. Also note that as w is still in the exit section, even if w' waits, the Unstoppable Writer property is not violated. After making sure that the previous writer has exited from $SWWP$, w' competes in $SWWP$ by executing its waiting room (Line 13). And once w' is in the critical section of $SWWP$, it enters the critical section (Line 13-14). If *W-token* $\notin \{0, 1\}$ at Line 11, then it means that the previous writer did not exit $SWWP$, so w' can enter the critical section without competing with the readers. Following theorem summarizes the properties of this algorithm.

Theorem 5 (Multi-Writer Multi-Reader lock with Writer Priority) *The algorithm in Figure 4 implements a Multi-Writer Multi-Reader lock satisfying the properties (P1)-(P6), (WP1) and (WP2) using the lock M from [3] and the algorithm in Figure 1. The RMR complexity of the algorithm in the CC model is $O(1)$. The algorithm uses $O(n)$ number of shared variables that support read, write, fetch&add and compare&swap operations.*

References

[1] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[2] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[3] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[4] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[5] Vassos Hadzilacos and Robert Danek. Local-spin group mutual exclusion algorithms. volume 3274, pages 71–85. Springer Berlin / Heidelberg, 2004.

[6] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[7] M. Raynal and D. Beeson. *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA, 1986.

[8] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003.

[9] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[10] Gary Granunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[11] James H. Anderson and Yong-Jik Kim. Fast and scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, London, UK, 1999. Springer-Verlag.

[12] Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion (extended abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 91–100, New York, NY, USA, 2000. ACM.

[13] James Anderson and Yong jik Kim. Adaptive mutual exclusion with local spinning. In *In Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, 2001.

[14] Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 295–304, New York, NY, USA, 2003. ACM.

[15] Jae heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:9–1, 1994.

[16] Danny Hendler and Philipp Woelfel. Randomized mutual exclusion in o(log n / log log n) rmrs. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 26–35, New York, NY, USA, 2009. ACM.

[17] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-time Systems Symposium*, pages 148–156. IEEE, 1993.

[18] Robert Cypher. The communication requirements of mutual exclusion. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 147–156, New York, NY, USA, 1995. ACM.

[19] Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 217–226, New York, NY, USA, 2008. ACM.

[20] James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. *Distrib. Comput.*, 15(4):221–253, 2002.

[21] Rui Fan and Nancy Lynch. An Ω (n log n) lower bound on the cost of mutual exclusion. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 275–284, New York, NY, USA, 2006. ACM.

[22] Vassos Hadzilacos. A note on group mutual exclusion. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 100–106, New York, NY, USA, 2001. ACM.

[23] Yuh-Jzer Joung. Asynchronous group mutual exclusion (extended abstract). In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 51–60, New York, NY, USA, 1998. ACM.

[24] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110, New York, NY, USA, 2009. ACM.

[25] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *In Proceedings of the 1993 International Conference on Parallel Processing*, pages 201–204. CRC Press, 1993.

[26] Bjorn B. Brandenburg and James H. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *ECRTS '09: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 184–193, Washington, DC, USA, 2009. IEEE Computer Society.

A Proofs for Section 3

A.1 Invariants of the algorithm in Figure 1

Our proof style is based on invariants, similar to Hoare style non-interference proofs. We identify various invariants which should be true when the writer w is about to execute a certain instruction. To prove the properties of the algorithm, first one has to prove the invariants itself and then use the invariants to prove the properties of the algorithm.

Now we present some notations used in the presentation of our invariants. We use $r(\textit{condition A})$ to denote the set or readers who would satisfy the *condition A*, e.g., $r(d = \overline{D},\ \text{PC} \in \{18 \dots 21\})$, denotes the set of readers that have their d value equal to $\overline{D}$ and have their program counter in the range $\{18 \dots 21\}$. $\mathcal{C}(d)$ refers to the set of readers who have incremented the *reader-count* component of $C(d)$ (line 17,20), but have not decrement it yet (line 22, 27). Similarly $\mathcal{EC}$ is defined.

The following proposition says that if a PC of a reader r is between line 18 and line 27, then r has incremented $C[d]$ (it might have incremented $C[\overline{d}]$ too), but has not decremented it yet.

Proposition A.1 *If r is a reader and $PC_r \in \{18, 27\}$ then $r \in C(\alpha)$ at time t, where α is the value of the d variable of r at t.*

The proposition is obvious from the code of `Read-lock()` in Figure 1. From now on we will take this proposition as a fact and not recall it every time the conditions of the proposition are true.

Some of the invariants of the algorithm in Figure 1 are given below. Now we describe one of those invariants to give some intuition behind them.

Lets take the invariant when $\text{PC}_w = 4$. The item 1 of the invariant just says what the value of $C[0], C[1]$ and *EC* would be in relation to $\mathcal{C}(0), C(1)$ and $\mathcal{EC}$ respectively. It also tells whether the writer has done an increment on *writer-waiting* component on any of these variables. As one can see from the code and the invariant that *writer-waiting* value of all of these variables should be 0 when $\text{PC}_w = 4$. The item 2 of the invariant just tells the state of the *Gate* variables. Item 3 says that all the reader who belong to $\mathcal{EC}$, i.e., have incremented *EC* but not decremented it, have their $d = \overline{D}$, i.e., belong to the previous side of the current writer. And those reader are not at line 28, this intuitively makes sense because the writer has not even incremented on the *writer-waiting* component of *EC*, hence has not asked the readers to wake it up through *ExitPermit*. Now lets look at the last item in the invariant, $r(d = D,\ \text{PC} \in \{23, 25 \dots 30\}) = \phi$. It says that there are no readers from the current side of the writer in the critical or exit section. This makes sense intuitively because the readers from the same side as the writer could not have entered the critical section as the writer made sure that the gate this side is closed already.

- When $\text{PC}_w \in \{1, 2, 3\}$, the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [0, |\mathcal{C}(\overline{D})|], EC = [0, |\mathcal{EC}|]$
  2. $\textit{Gate}[D] = \textit{true}, \textit{Gate}[\overline{D}] = f$
  3. $\mathcal{EC} = r(d = D,\ \text{PC} \in \{27, 29\})$
  4. $r(d = D,\ \text{PC} \in \{28, 30\}) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18..27\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18..21\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
  7. $r(d = \overline{D},\ \text{PC} \in \{22..30\}) = \phi$
- When $\text{PC}_w = 4$, the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [0, |\mathcal{C}(\overline{D})|], EC = [0, |\mathcal{EC}|]$
  2. $\textit{Gate}[D] = \textit{false}, \textit{Gate}[\overline{D}] = \textit{true}$
  3. $\mathcal{EC} = r(d = \overline{D},\ \text{PC} \in \{27, 29\})$
  4. $r(d = \overline{D},\ \text{PC} \in \{28, 30\}) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 22, 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 27\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
  7. $r(d = D,\ \text{PC} \in \{23, 25 \dots 30\}) = \phi$
- When $\text{PC}_w = 5$, the following hold:
  Same as when $\text{PC}_w = 4$ except $\textit{Permit}[\overline{D}] = \textit{false}$

- When $\text{PC}_w = 6$ and *Permit*$[\overline{D}]$ = *false*, the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [1, |\mathcal{C}(\overline{D})|], EC = [0, |\mathcal{EC}|]$
  2. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  3. $\mathcal{EC} = r(d = \overline{D},\ \text{PC} \in \{27 \dots 29\})$
  4. $|r(d = \overline{D},\ \text{PC} = 28)I \leq 1$ and $r(d = \overline{D},\ \text{PC} = 30) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)]$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 27\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
  7. $|\mathcal{C}(\overline{D})| + r(\ \text{PC} \in \{23, 28\}) > 0$
  8. $r(d = D,\ \text{PC} \in \{25 \dots 30\}) = \phi$
- When $\text{PC}_w = 6$ and *Permit*$[\overline{D}]$ = *true*, the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [1, |\mathcal{C}(\overline{D})|], EC = [0, |\mathcal{EC}|]$
  2. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  3. $\mathcal{EC} = r(d = \overline{D},\ \text{PC} \in \{28 \dots 29\})$
  4. $|r(d = \overline{D},\ \text{PC} = 28)| \leq 1$ and $r(d = \overline{D},\ \text{PC} = 30) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 21\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
  7. $r(d = D,\ \text{PC} \in \{25 \dots 30\}) = \phi$
- When $\text{PC}_w = 7$, the following hold:
  Same as for the case $\text{PC}_w = 6$ and *Permit*$[\overline{D}]$ = *true*.
- When $\text{PC}_w = 8$, the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [0, |\mathcal{C}(\overline{D})|], EC = [0, |\mathcal{EC}|]$
  2. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  3. $\mathcal{EC} = r(d = \overline{D},\ \text{PC} \in \{28 \dots 29\})$
  4. $|r(d = \overline{D},\ \text{PC} = 28)| \leq 1$ and $r(d = \overline{D},\ \text{PC} = 30) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 21\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
  7. $r(d = D,\ \text{PC} \in \{25 \dots 30\}) = \phi$
- When $\text{PC}_w = 9$, the following hold:
  Same as when $\text{PC}_w = 8$ except that *Gate*$[\overline{D}]$ = *false*
- When $\text{PC}_w = 10$, the following hold:
  Same as when $\text{PC}_w = 9$ except that *ExitPermit* = *false*
- When $\text{PC}_w = 11$ and *ExitPermit* = *false* , the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [0, |\mathcal{C}(\overline{D})|], EC = [1, |\mathcal{EC}|]$
  2. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *false*
  3. $\mathcal{EC} = r(d = \overline{D},\ \text{PC} \in \{28 \dots 29\})$
  4. $|r(d = \overline{D},\ \text{PC} \in \{28, 30\})| \leq 1$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 21\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
  7. $\mathcal{EC} \cup r(\ \text{PC} = \{30\}) \neq \phi$
  8. $r(d = D,\ \text{PC} \in \{25 \dots 30\}) = \phi$
- When $\text{PC}_w = 11$ and *ExitPermit* = *true* , the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [0, |\mathcal{C}(\overline{D})|], EC = [1, |\mathcal{EC}|]$
  2. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *false*
  3. $\mathcal{EC} = \phi$
  4. $r(\ \text{PC} \in \{25 \dots 30\}) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 21\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$
- When $\text{PC}_w = 12$, the following hold:
  Same as the case when $\text{PC}_w = 11$ and *ExitPermit* = *true*
- When $\text{PC}_w = \in \{13, 14\}$, the following hold:
  1. $C[D] = [0, |\mathcal{C}(D)|], C[\overline{D}] = [0, |\mathcal{C}(\overline{D})|], EC = [0, |\mathcal{EC}|]$
  2. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  3. $\mathcal{EC} = \phi$
  4. $r(\ \text{PC} \in \{25 \dots 30\}) = \phi$
  5. $\mathcal{C}(D) = r(d = D,\ \text{PC} \in \{18 \dots 24\}) \cup$ $r(d = \overline{D}, d' = D,\ \text{PC} = 21)$
  6. $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \dots 21\}) \cup$ $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$

A.2 Proof of Theorem 1

Based on the invariants given above we prove properties of the algorithm in Figure 2.

Lemma 6 (Mutual Exclusion) *A reader r and the writer w cannot be in the critical section together.*

PROOF. This lemma is obvious from the invariant when $\text{PC}_w = 13$. □

Lemma 7 (Writer Priority) *If the writer w has better priority than a reader r then, r does not enter critical section before w.*

PROOF. One can see that when there is only one writer in the system, if w has better priority than r, it means that w doorway precedes r. In the other words r was in remainder section when $\text{PC}_w = 4$. The following lemma states that such an r would never set its d to $\overline{D}$ while w is in try section.

Lemma 8 *If w is in try section and doorway precedes some reader r in try section, then d of r is not equal to $\overline{\text{D}}$ while w is active.*

As the writer only updates *D* at line 3 and r starts its try section after w has executed line 3, so r would set its $d = D$ at line 16. From the rest of the code for `Read-lock`, one can see that r changes its d variable only if observes some change in *D*, this is not possible while w is active. □

To see why the lemma holds, observe that value $Gate[d]$ is *false* when $\text{PC}_w \in [4, 14]$. This clearly means that r cannot go past line 24 and hence cannot enter the critical section while w is still in try section. □

Before proving the Unstoppable Writer property, we define what we mean by a process not occupying the critical section.

Definition 5 (process not occupying critical section) *A process p from an attempt A does not occupy the critical section, if either the A has completed or p is in exit section in A and some process with conflicting p has entered the critical section after p.*

Lemma 9 (Unstoppable Writer Property) *If the writer w is in the waiting room, w has better priority than all the readers in the try section and no reader occupies CS then w is CS-enabled.*

PROOF. First we claim that if a reader does not occupy *CS* then it cannot be in critical or exit section. Say a reader r does not occupy *CS* , by definition it cannot be in critical section. And if it is in some exit section then some writer before w, say w', entered the critical section after r. If we look at the invariants when PC of w' is 13, one can clearly see that no process is in the exit section ($\mathcal{EC} = \phi$). Hence r cannot be still in the exit section. This fact together with the following lemma proves the lemma.

Lemma 10 *If the writer w is in the waiting room, w doorway precedes all the readers in try section, and no readers are in critical or exit section then w is CS-enabled.*

PROOF. By lemma 8 all the no reader in try section will have $d = \overline{D}$. Also as w doorway precedes all the readers in try section, there will be no reader in the try section with $d' = \overline{D}$.

Suppose, w is not *CS* -enabled, that means it will spin at the loops at line 6 or 11. Lets consider both the cases.

Case (a) w spins at line 6 : this means that it finds $Permit[\overline{D}]$ to *false* when $\text{PC}_w = 6$. By item 6 of the invariant when $\text{PC}_w = 6$ and $Permit[D] = false$, $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \ldots 27\}) \cup r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$. As we noticed earlier no reader in try section has $d' = \overline{D}$, so $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\}) = \phi$. And as there are no readers in try section or critical section with $d = \overline{D}$ so $\mathcal{C}(\overline{D}) = \phi$. By the item 7 of the invariant for $\text{PC}_w = 6$ and $Permit[D] = false$, one can see that

$|\mathcal{C}(\overline{D})| + r(\text{ PC} \in \{23, 28\}) > 0$, but this is a contradiction as there are no readers in $\mathcal{C}(\overline{D})$ and the exit section.

Case (b) w spins at line 11 : this means that it finds *ExitPermit*$[\overline{D}]$ to *false*. By the item 7 of the invariant when $\text{PC}_w = 11$ and *ExitPermit* = *false*, $\mathcal{EC} \cup r(\text{ PC} = \{30\}) \neq \phi$. But this is a clear contradiction to the fact that there are no readers in exit section.

□

□

Lemma 11 (Starvation Freedom) *If no process stays in critical section forever then any process which enters the try section eventually enters the critical section.*

PROOF. We prove this lemma by covering different cases when the writer w or a reader r is in try section forever.

Case (a) : w spins at line 6 forever : Then we prove the following claim that $\mathcal{C}(D)$ will eventually be empty forever.

Claim 11.1 *If w spins at line 6 forever, then eventually $\mathcal{C}(D)$ will be empty forever.*

PROOF. w spins at line 6 forever. It means that *Permit*$[\overline{D}]$ is equal to *false* forever. So if we look at the invariants when $\text{PC}_w = 6$ and *Permit*$[\overline{D}]$ = *false*, we get $\mathcal{C}(\overline{D}) = r(d = \overline{D},\ \text{PC} \in \{18 \ldots 27\}) \cup r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$.

First we claim that $r(d = \overline{D},\ \text{PC} \in \{18 \ldots 27\})$ will eventually be empty. So see why this is true say some reader has $d = \overline{D}$, one can see from the invariants for $\text{PC}_w = 5$, that *Gate*$[\overline{D}]$ = *true*, hence this reader will eventually either enter the critical section or update its d to D at line 21. Also any reader which will update its d variable while $\text{PC}_w = 5$ cannot update it to $\overline{D}$. Using the previous two facts and the assumption that no process stays in critical or exit section one can see that $r(d = \overline{D},\ \text{PC} \in \{18 \ldots 27\})$ will eventually be empty.

Now to show that eventually $\mathcal{C}(\overline{D})$ is empty, we have to show that eventually $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\})$ is empty. Note that eventually any reader who updates d' (line 18) will set it to D and not to $\overline{D}$, hence there will be a point when no more readers are added to this set. Also, all the readers in this set will eventually go past line 22 and will be removed form this set. Hence this proves that eventually forever $r(d = D, d' = \overline{D},\ \text{PC} \in \{21, 22\}) = \phi$. □

Now from the invariant 7 for $\text{PC}_w = 5$ and *Permit*$[\overline{D}]$ is equal to *false*, one can see that when $\mathcal{C}(\overline{D}) = \phi$, there has to be some process at line 23 or 28. In both these lines a reader sets *Permit*$[\overline{D}]$ to *true*. Hence eventually when a reader executes either of these lines, *Permit*$[\overline{D}]$ = *true* and the writer does not spin on line forever, a contradiction.

Case (b) w spins at line 11 forever. Exactly like the the previous case with the difference that now one has to argue that eventually $\mathcal{EC}$ will be empty forever.

Hence we have shown that the writer w never starves. To show that the reader never starves we prove the following lemma first.

Lemma 12 *If a reader r is in the waiting room and $PC_w \in \{1 \dots 3\}$ then r is CS-enabled.*

PROOF. Say r is in the waiting room ($PC_r = 24$), and w is in the remainder section ($PC_w = 1$). Then by the item 7 of the invariant when $PC_w = 1$, one can see that d of r is equal to D, say $d = 1$. By the item 2 of the same invariant one can see that $Gate[1] = true$. We prove the lemma by making the following claim that proves that w does not set $Gate[1]$ to *false* while r is at line 24.

Claim 12.1 *If a reader r is at line 24 with $d = \alpha, \alpha \in \{0, 1\}$ and* Gate$[\alpha]$ *is equal to true, then* Gate$[\alpha]$ *is not set to false while r is at line 24.*

PROOF. Suppose w does set $Gate[\alpha]$ from *true* to *false* (at line 8) while r is still at line 24. If we look at the invariants when $PC_w = 8$, only $Gate[\overline{D}] = true$ (item 2 of the invariant when $PC_w = 8$), hence $\alpha = \overline{D}$. And all the processes at line 24 have $d = D = \overline{\alpha}$ (item 5,6), but this contradicts the fact that r is at line 24 with $d = \alpha$. □

□

Case (c) r spins at line 24 forever. If the writer ever enters the remainder section while r is at line 24, by the lemma 12, r should be enabled. Now if w is in try section, then by case (a) and (b) of this lemma, it will eventually enter the critical section, and by assumptions of the lemma eventually exit and go to the remainder section. Hence in ether case, r will be eventually *CS* -enabled, hence cannot spin at line 24 forever.

□

Lemma 13 (Concurrent Entry) *If a reader r is in the waiting room, and the writer is not in try section and does not occupy CS then r should be enabled.*

PROOF. We claim that if the writer is not in try section and does not occupy *CS* then it should be in remainder section, this together with lemma 12 will prove the lemma. If w does not occupy *CS* then by definition it cannot be in the critical section. And if its in exit section then some reader entered the critical section after it. But as there is only one statement in the exit section of w, line 14. By looking at the invariants when $PC_w = 14$, one can clearly see that no reader should be in critical section, it means that w has to completely leave the exit section to not occupy *CS* .

□

Lemma 14 (First in First Enabled) *If a reader r is in the waiting room, another reader r' is in the critical section and r doorway precedes r' then r should be CS-enabled.*

PROOF. As r is in the waiting room so $PC_r = 24$, W.L.O.G say d of r is equal to 1. So r is spinning till $Gate[1] = true$. If d of r' is also equal to 1, it means that r' saw $Gate[1] = true$ at some point when r was at line 24. By using claim 12.1 one can see that r should also be enabled in this case.

Now suppose d of r is 1 and d of r' is 0. If $PC_w \in \{1 \dots 3\}$ at the point when r executes line 24 for the first time, then by lemma 12, r should be enabled. So lets, suppose $PC_w > 3$ and $Gate[1] = false$ when r executes

line 24 for the first time, say at time t. One can see from the item 2 of the invariants for $\text{PC}_w > 3$ that $D = 1$ at t. But as r' starts its doorway after r and set its d equal to 0 at some time while r is at line 24. It means that the writer entered the remainder section while r was at line 24, hence by lemma 12 again r should be enabled. □

The following property was not given in the main part of the paper, but we claim that it is very useful and our algorithm satisfies it.

Lemma 15 (Waiting Reading Enabled property) *If a reader r is in the waiting room, the writer w is in critical section, then r should be CS-enabled at the time some reader enters critical section after w.*

PROOF. Say r enters the waiting room at time t and let r' be the earliest reader to enter *CS* after w, and say it enters *CS* at time t'. As w is in critical section at some time in the interval $[t, t')$, and r' is in *CS* at t', it means that w exited *CS* at some time in the interval $[t, t')$. By the invariants when $\text{PC}_w = 14$, w cannot be at line 14 when, r' enters *CS* . Hence this means that w enters the remainder section at some time in the interval $[t, t')$, hence r should be *CS* -enabled at t'. □

Lemma 16 (Constant RMR complexity) *The algorithm given in Figure 1 has $O(1)$ RMR complexity in CC model.*

PROOF. As both `Write-lock()` and `Read-lock()` have constant number of steps, all we have to argue is that the RMR complexity of their *wait till* statement is also constant. It is very easy to see that for `Write-lock()`. As the writer waits for *Permit*$[\overline{D}]$ and *ExitPermit* to be set to *true* at line 6 and 11 respectively, and as writer is the only process to set them to *false*, so value of these variables only changes once while the writer is spinning on them.

In a `Read-lock()`, a reader waits till *Gate*$[d]$ is set to true (line 24). By claim 12.1, once the value of *Gate*$[d]$ is set to *true*, it is not set back to *false* while the reader is still at line 24. Hence `Read-lock()` also has $O(1)$ RMR complexity in CC model. □

B Proofs from section 4

B.1 Invariants of the Algorithm in Figure 2

The invariants for the algorithm in Figure 2 are presented in Figure 5. The notations used here are similar to the notations used in previous section. Using the invariants given in 5, here we present the proofs of the properties the algorithm in Figure 2 satisfies.

B.2 Proof of Theorem 2

Lemma 17 (Mutual Exclusion) *A reader r and the writer w cannot be in the critical section together.*

PROOF. One can easily see from invariant for $\text{PC}_w = 6$ that there is no reader r in critical section ($\text{PC}_r = 25$). □

One can see that **Concurrent entry** (P5) and **FIFE** (P4) are mere consequence or the Unstoppable Reader Property. Also as there is only one writer, Unstoppable Reader Property also implies **Reader Priority** (RP1). Now we prove Unstoppable Reader Property.

Global invariant : $C = |r\{\ \text{PC} \in 15 \ldots 22\}|$

- $\text{PC}_w = 1, 2$
  1. *Gate*$[D]$ = *true*, *Gate*$[\overline{D}]$ = *false*
  2. $X \neq$ *true*, $r(\mathcal{PID} \in X,\ \text{PC} \in \{14, 15\}) = \phi$
  3. *Permit* = *true*
  4. $r(d = \overline{D},\ \text{PC}\{20 \ldots 26\}) = r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w = 3$
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $X \neq$ *true*, $r(\mathcal{PID} \in X,\ \text{PC} \in \{14, 15\}) = \phi$
  3. *Permit* = *true*
  4. $r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w \in [10 - 15]$, $X \neq$ *true*
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $r(\ \text{PC} = 16) = \phi$
  3. *Permit* = *false*
- $\text{PC}_w = [10 - 15]$, $X =$ *true* and *Permit* = *false*
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $r(d = \overline{D},\ \text{PC} \in \{20 \ldots 26\}) = r(d = D,\ \text{PC} \in \{25, 26\}) = \phi$
  3. $|r(\ \text{PC} = 16)| = 1$
- $\text{PC}_w = [10 - 15]$, $X =$ *true* and *Permit* = *true*
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $r(d = \overline{D},\ \text{PC} \in \{20 \ldots 26\}) = r(d = D,\ \text{PC} \in \{25, 26\}) = \phi$
  3. $r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w = 16$
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $X =$ *true*
  3. *Permit* = *false*
  4. $r(d = \overline{D},\ \text{PC} \in \{20 \ldots 26\}) = r(d = D,\ \text{PC} \in \{25, 26\}) = \phi$
  5. $r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w = 5$ and $X \in \mathcal{PID}$
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $|r(\ \text{PC} \in \{19 \ldots 27, 10\})| + |r(x = X,\ \text{PC} \in \{11, 12\})| + |r(\mathcal{PID} = X,\ \text{PC} \in [13, 15])| > 0$
  3. *Permit* = *false*
  4. $r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w = 5$, $X =$ *true*, *Permit* = *false*
  Same as when $\text{PC}_w = [10 - 15]$, $X =$ *true* and *Permit* = *false*
- $\text{PC}_w = 5$, $X =$ *true*, *Permit* = *true*
  Same as when $\text{PC}_w = [10 - 15]$, $X =$ *true* and *Permit* = *true*
- $\text{PC}_w = 6, 7$
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *true*
  2. $X =$ *true*
  3. $r(d = \overline{D},\ \text{PC} \in \{20 \ldots 26\}) = r(d = D,\ \text{PC} \in \{25, 26\}) = \phi$
  4. *Permit* = *true*
  5. $r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w = 8$
  1. *Gate*$[D]$ = *false*, *Gate*$[\overline{D}]$ = *false*
  2. $X =$ *true*
  3. $r(d = \overline{D},\ \text{PC} \in \{20 \ldots 26\}) = r(d = D,\ \text{PC} \in \{25, 26\}) = \phi$
  4. *Permit* = *true*
  5. $r(\ \text{PC} = 16) = \phi$
- $\text{PC}_w = 9$
  1. *Gate*$[D]$ = *true*, *Gate*$[\overline{D}]$ = *false*
  2. $X =$ *true*
  3. $r(d = \overline{D},\ \text{PC} \in \{20 \ldots 26\}) = \phi$
  4. *Permit* = *true*
  5. $r(\ \text{PC} = 16) = \phi$

Figure 5: Invariants for algorithm in Figure 2

Lemma 18 (Unstoppable Reader Property) *If a reader r is in the waiting room (line 24), r has better priority than the writer w (if w is in try section), and the writer does not occupy* CS *then r is* CS *-enabled.*

PROOF. First we prove the following two lemmas which are used to proving the lemma.

Lemma 19 *If a reader r in the waiting room and $PC_w \in \{1, 9\}$ then r is CS-enabled.*

PROOF.

Lets take the case of $\text{PC}_w = 1$. As r is in the waiting room it means $\text{PC}_r = 24$. So one can see from the invariants when $\text{PC}_w = 1$ that variable d of r is equal to *D* and $Gate[d] = true$. WLOG let $d = D = 1$, we will show that $Gate[1]$ is not set to *false* while r is at line 24.

Say $Gate[1]$ is set to *false* later while r is still at line 24. It means w executes $Gate[1] \leftarrow false$ (line 7), but by the invariant for $\text{PC}_w = 7$, $r(d = 1,\ \text{PC} \in \{20 \dots 26\}) = \phi$, this contradicts the fact that r is still at line 24. The case for $\text{PC}_w = 9$ is exactly the same. □

Lemma 20 *If at time t a reader r is in the waiting room and some reader is in critical section then r is CS-enabled.*

PROOF. If at time t, $X = true$ and some reader is in critical section (line 25), from the invariants one can clearly see that $\text{PC}_w = 9$, and from the lemma 19, r should be *CS* -enabled.

As $\text{PC}_r = 24$ it means r previously observed $X = true$ at line 23. So if $X \neq true$ at time t, by the inspection of the code once can see that only the writer changes X when it is *true* (line 9). Hence it means w executed line 9 during the interval $r[23, t]$. Hence by the lemma 19, r should be *CS* -enabled. □

To prove the lemma lets first consider the case when the writer is not in try section. As the writer w does not occupy *CS* , then it either is in the remainder section ($\text{PC}_w = 1$) in which case using lemma 19 we are done, or w is in exit section and some other reader entered the *CS* while w is in exit section ($\text{PC}_w \in [7, 9]$). By inspecting at the invariants when $\text{PC}_w \in [7, 9]$, one can clearly see that for some reader to enter *CS* while w is in exit section, w has to be at line 9. Hence by using lemma 19, r should be *CS* -enabled.

Now lets consider the case when the writer is in the try section. As the reader has better priority than the writer w then by the definition of priority either w was in remainder section while r is in waiting room, or some reader was in *CS* while r is in waiting room. By using the lemma 19 for the former case and lemma 20 for the later case we can see that r should be enabled. □

Lemma 21 (Reader Starvation freedom) *If a reader r is in try section and the writer does not stay in critical section forever then r eventually enters the critical section.*

PROOF. Suppose r stays in try section forever. Then we claim that the writer w also stays in the try section forever. This is true because, if r keeps taking steps then it will eventually complete its doorway. Now if w ever enters remainder section, by lemma 19, r will be *CS* -enabled. If w enters the critical section, by assumption of the lemma it will have to exit and enter remainder eventually (as exit section is bounded too). So one can see both r and w will be in the try section forever, which means $\text{PC}_r = 24$, $\text{PC}_w = 5$ and $Permit = false$ forever.

Now we claim that $X = true$ while r is at line 24. So see why this is true, note that as r is at line 24, it would have seen $X = true$ at line 23. But if $X \neq true$ later while r is at line 24, it means that w would have executed line 9 while r is in waiting room. Hence by lemma 19, r should be *CS* enabled which is a contradiction.

So which means $\text{PC}_w = 5, X = \textit{true}$ and $\textit{Permit} = \textit{false}$, but looking at the invariant for this case, one can see that some reader is at line 16 and it will eventually execute it, which means that *Permit* would be set to *true* and w will enter *CS* eventually which is a contradiction. □

Lemma 22 (Livelock freedom) *If some process is in the try section and no process is in the critical and exit section then some process enters the critical section eventually.*

PROOF. As we have already shown starvation freedom for readers all we have to show is that if no reader is active then the writer does not starve forever.

Say the writer stays in try section forever, so $\text{PC}_w = 5$ and $\textit{Permit} = \textit{false}$ forever. If $X = \textit{true}$ then by the invariants when $\text{PC}_w = 5$, $X = \textit{true}, \textit{Permit} = \textit{false}$, one can see that there is one reader at line 16, which is a contradiction. If $X \neq \textit{true}$, then by the item 3 of the invariant when $\text{PC}_w = 5$, $X \neq \textit{true}, \textit{Permit} = \textit{false}$, one can see that some reader is active, which is a contradiction. □

Lemma 23 (Constant RMR complexity) *The algorithm given in Figure 2 has $O(1)$ RMR complexity in CC model.*

PROOF. As the doorway and exit section of both `Write-lock()` and `Read-lock()` have constant number of steps, all we have to show is that the RMR complexity when the processes are spinning is constant.

In case of `Write-lock` it is easy to see why this is true ; when the *Permit* is set to *true* in the waiting room of the writer, it is *CS* -enabled. As writer is the only process to set *Permit* to *false*, so *Permit* would remain *true* till the writer goes to the critical section.

In the `Read-lock`, say a reader r is in the waiting room (line 24), say without loss of generality, r is waiting for $\textit{Gate}[1]$ to be set to *true*, we claim that once $\textit{Gate}[1]$ is equal to *true*, it is not set back to *false* while r is still at line 24. To see why this claim is true notice that only the writer ever writes into the *Gate* variable and it only sets it to false at line 7. Say $\textit{Gate}[1]$ is *true* at this point and $\text{PC}_w = 7$, by the invariants when $\text{PC}_w = 7$ one can see that there should be no reader at line 24 with $d = 1$, but this contradicts the assumption that r is at line 24 waiting for $\textit{Gate}[1]$ to be set to *true*. Now with this claim one can see that RMR complexity of `Read-lock` in CC model is also constant. □