Optimal Memory-Anonymous Symmetric Deadlock-Free Mutual Exclusion

Zahra Aghazadeh, Damien Imbs, Michel Raynal, Gadi Taubenfeld, Philipp Woelfel

▶ **To cite this version:**

Optimal Memory-Anonymous
Symmetric Deadlock-Free Mutual Exclusion

Zahra Aghazadeh[1], Damien Imbs[2] Michel Raynal[3,4] Gadi Taubenfeld[5] Philipp Woelfel[1]

[1] Dept of Computer Science, University of Calgary
[2] LIS, Université d'Aix-Marseille, CNRS, Université de Toulon, Marseille, France
[3] Univ Rennes IRISA, 35042 Rennes, France
[4] Department of Computing, Polytechnic University, Hong Kong
[5] The Interdisciplinary Center, Herzliya, Israel

Abstract—The notion of an anonymous shared memory, introduced by Taubenfeld in PODC 2017, considers that processes use different names for the same memory location. As an example, a location name A used by a process p and a location name $B \neq A$ used by another process q can correspond to the very same memory location X, and similarly for the names B used by p and A used by q which may (or may not) correspond to the same memory location $Y \neq X$. In this context, the PODC paper presented a 2-process symmetric deadlock-free mutual exclusion (mutex) algorithm and a necessary condition on the size m of the anonymous memory for the existence of such an n-process algorithm. This condition states that m must be belongs to $M(n) \setminus \{1\}$ where $M(n) = \{m : \forall \ell : 1 < \ell \leq n : \gcd(\ell, m) = 1\}$. *Symmetric* means here that, process identities define a specific data type which allows a process to check only if two identities are equal or not.

The present paper presents two optimal deadlock-free symmetric mutual exclusion algorithms for n-process systems where communication is through m registers. The first algorithm, which considers anonymous read/write registers, works for any m which is $\geq n$ and belongs to the set $M(n)$. It follows that this condition on m is both necessary and sufficient for symmetric deadlock-free mutual exclusion in this anonymity context, and this algorithm is optimal with respect to m. The second algorithm, which considers anonymous read/modify/write atomic registers, works for any $m \in M(n)$, which is shown to be necessary and sufficient for anonymous read/modify/write registers. It follows that, when $m > 1$, $m \in M(n)$ is a tight characterization of the size of the anonymous shared memory needed to solve deadlock-free mutex, be the registers read/write or read/modify/write.

KEYWORDS: Anonymous shared memory, Asynchronous system, Atomic register, Computability, Deadlock-freedom, Mutual exclusion, Read/modify/write register, Read/write register.

I. INTRODUCTION

A. Memory Anonymity

While the notion of *memory-anonymity* has been implicitly used in some works in the early eighties (e.g., [16]), it was explicitly captured as a concept and investigated only very recently in [21][1]. More precisely, this paper considers the explicit case where "there is no a priori agreement between processes on the names of shared memory locations", and presents possibility/impossibility results in such systems.

Considering a shared memory defined as an array $R[1..m]$ of m memory locations (registers), memory-anonymity means that, while the same location identifier $R[x]$ always denotes the same memory location for a process p, it does not necessarily denote the same memory location for two different processes p and q. This means that there is an adversary that may initially give different global names to different processes for the same memory locations. In other words, the adversary associates a permutation on the set of memory indices $\{1, \ldots, m\}$ with each process p, and p uses them to access the memory locations. This is illustrated in Table I, which considers two processes p and q and a shared memory comprising three registers (as an example the register known as $R[2]$ by p and the register known as $R[3]$ for q are the very same register, which actually is $R[1]$ from an external omniscient observer point of view).

names for an external observer	location names for process p	location names for process q
$R[1]$	$R[2]$	$R[3]$
$R[2]$	$R[3]$	$R[1]$
$R[3]$	$R[1]$	$R[2]$
permutation	$2, 3, 1$	$3, 1, 2$

Table I
EXAMPLE OF AN ANONYMOUS MEMORY MODEL

B. Related Work and Motivation

Related work. The memory-anonymous mutual exclusion problem was introduced by Taubenfeld in [21], where are

---

[1]See [18] for an introductory survey to process anonymity and memory anonymity.

presented the first results on mutual exclusion in read/write memory-anonymous systems. More precisely, are presented

- a symmetric deadlock-free algorithm for two processes (the notion of "symmetric" is related to the use of process identifiers, and will be defined in Section II-C);
- a theorem stating that there is no deadlock-free algorithm if the number of processes is not known; and
- a necessary condition stating that any deadlock-free symmetric mutex algorithm for n processes, which uses $m \geq 2$ read/write registers, requires that m belongs to the set $M(n) = \{m : \forall \ell : 1 < \ell \leq n : \gcd(\ell, m) = 1\}$ (i.e., ℓ and m are relatively prime). Let us observe that $M(n)$ is infinite (among other values, it contains an infinite number of prime numbers).

A common technique employed by shared memory algorithms that aims at minimizing the number of read/write registers, is that a process keeps writing its ID (and possibly additional information) to different registers, until sufficiently many registers store the information written by that process. Sufficiently many registers means that, even if all other processes write at the same time (possibly to different registers), they cannot obliterate too much information. This idea is used, e.g., by obstruction-free algorithms for space efficient test-and-set objects [7], [8] (which can be viewed as one-time mutual exclusion primitives). Our mutual exclusion algorithms also require a process to "win" sufficiently many registers by writing its ID to them. However, this serves another purpose, in addition to ensuring that not too much information can get obliterated by overwrites: Once all $m \in M(n)$ registers have been written, not every process's ID can appear equally often, which allows breaking ties. The main algorithmic idea is to let processes "remove" themselves from registers, if the tie-breaking mechanism favors other processes. This has to be done carefully to ensure that different processes, which may view different system configurations, do not all back off at the same time.

Motivation: This work has two primary motivations. The first is related to the basics of computing, namely, computability and complexity lower/upper bounds. Increasing our knowledge of what can (or cannot) be done in the context of an anonymous memory, and providing associated necessary and sufficient conditions, helps us determine the weakest possible system assumptions under which fundamental problems, such as mutual exclusion, can be solved. The second one is more application-oriented. It appears that the concept of an anonymous memory allows epigenetic cell modification to be modeled from a computing point of view [19]. Hence, anonymous shared memories could be useful in biologically inspired distributed systems [14], [15]. If this is the case, mastering mutex in such an adversarial context could be of primary importance.

C. Content of the Paper

As announced, this paper is on deadlock-free mutual exclusion in an n-process system ($n \geq 2$), where the processes communicate by accessing a shared memory composed of m anonymous registers (hence there are no other non-anonymous registers).

Preliminary definitions: Two types of registers are considered, giving rise to two communication models.

- A read/write (RW) register is an atomic register that provides the processes with a read operation and a write operation [12]. In the RW communication model, all registers are RW registers.
- A read/modify/write (RMW) register is an atomic register that provides the processes with a read operation, a write operation, and an additional read/modify/write operation [10]. Such an operation allows a process to atomically read a register, and, based on the value read, compute a new value that is assigned to the register. One of the most prominent read/modify/write operations is compare&swap(). When a process invokes R.compare&swap(x, old, new), where $R[x]$ is an anonymous register, it reads the value of the register locally known as $R[x]$, say v, and assigns it the value new if and only if $new = v$. In that case the operation returns true, and otherwise it returns false. In the RMW communication model, all the registers are RMW registers.

Results presented in the paper:

- RW anonymous communication model. A deadlock-free mutex algorithm for this model is presented, which assumes the necessary condition stated above, namely, $m \in M(n)$. As the condition $m \geq n$ is necessary to solve mutex in an RW non-anonymous system [3], it remains necessary in an anonymous system. The very existence of the proposed algorithm shows that the predicate $m \in M(n) \wedge m \geq n$ (which is equivalent to $m \in M(n) \wedge m \neq 1$) is a tight characterization of the values of m that permit a deadlock-free mutex algorithm in RW anonymous systems. In this sense, the proposed algorithm is space optimal.
- RMW anonymous communication model. A deadlock-free mutex algorithm for this model is presented, which requires $m \in M(n)$. As, in this communication model, $m \geq n$ is not a previously known necessary requirement to solve deadlock-free mutex in a non-anonymous system, we cannot conclude from the existence of the previous algorithm that $m \in M(n)$ is a tight characterization of the values of m which allow deadlock-free mutex algorithm in RMW anonymous systems.

  To address this issue, the paper presents a proof that $m \in M(n)$ is actually a necessary and sufficient condition for deadlock-free mutex in RMW anonymous systems. Let us observe that when the whole memory is

composed of a single register (case $m = 1$), the fact that it is anonymous or not is irrelevant as when they access the shared memory, the processes access the very same register. (Said differently, when $m = 1$, the adversary defining the addressing permutations has no power, and consequently the notion of non-anonymous/anonymous register disappears.)

Hence, eliminating the pathological case $m = 1$, deadlock-free mutex for both the RW model and the RMW model can be solved if and only if $m \in M(n)$. The corresponding algorithms differ in their algorithmic design, and in the fact that, to enter the critical section, the algorithm for the RW model requires a process to read its identity from all m anonymous registers, while the algorithm for the RMW model requires it to read its identity from a majority of the anonymous registers only. Hence, while requiring the same computability assumption (namely $m \in M(n)$ assuming $m \neq 1$), these algorithms differ from a complexity point of view (measured as the number of registers that have to contain the same process identity to allow this process to enter the critical section).

D. Roadmap

This paper is composed of VII sections. Section II introduces the computational model and provides some technical definitions. Section III presents an n-process symmetric deadlock-free mutex algorithm for the anonymous RW communication model. Section IV proves its correctness. Section V presents an n-process symmetric deadlock-free mutex algorithm for the anonymous RMW communication model. Section VI-A proves its correctness, while Section VI-B proves a space lower bound which shows that the algorithm is space optimal. Finally, Section VII concludes the paper. It is important to notice that both algorithms have a first class noteworthy property, namely, their simplicity.

II. SYSTEM MODEL, SYMMETRIC ALGORITHM, AND MUTUAL EXCLUSION

A. Processes and Anonymous Registers

The system is composed of n asynchronous processes p_1, ..., p_n with identifiers from a set $\mathcal{P}$. "Asynchronous" means that each process proceeds in its own speed, which can vary with time and always remains unknown to the processes.

When considering a process p_i, i is its index, which is used only to distinguish processes from an external point of view. A process p_i knows its own identity, denoted id_i, but never knows its index i. No two processes have the same identity. Each process knows the number n of processes in the system, and all processes know a common symbol $\bot \notin \mathcal{P}$, which is interpreted as a default identity (hence, when it reads an anonymous register, a process can distinguish a process identity from $\bot$).

B. Anonymous Registers

RW communication model: Processes communicate through a shared anonymous memory, represented by an array $R[1..m]$. It is important to stress that the size m of the shared memory is imposed by the system model, and is not under the control of the programmer. The entire memory is anonymous. Consequently the only means for processes to communicate is by accessing $R[1..m]$. Each register can be accessed by two operations, denoted write and read. As already indicated, memory-anonymity means that, for each process p_i, there is a permutation $f_i()$ over the set $\{1, \ldots, m\}$ such that, when p_i uses address $R[x]$, it actually accesses $R[f_i(x)]$. Anonymity means that no permutation f_i is known by any process; each f_i is defined by an adversary.

- When a process p_i invokes $R.\text{write}(x, v)$ it writes value v in the atomic read/write register $R[f_i(x)]$,
- When a process p_i invokes $R.\text{read}(x)$ it obtains the value currently saved in the register locally denoted $R[f_i(x)]$.

To simplify the presentation of the first algorithm, we also assume that process p_i can use an operation $R.\text{snapshot}()$ to obtain the value of array $\big[R[f_i(1)], \ldots, [R[f_i(m)]]\big]$ as if the read of all its entries were instantaneous (i.e., produced at a single point of the time line during the operation [10], [12]). We require that the operation $\text{snapshot}()$ satisfies the following progress guarantee (slightly modified from [9]):

> Any invocation of $R.\text{snapshot}()$ by a process p_i terminates within a finite number of p_i's steps, given that no process invokes $\text{write}()$ during its execution. $\quad$ (1)

The memory-anonymous $\text{snapshot}()$ operation is a simple extension of the classical $\text{snapshot}()$ introduced in [1], [2]. All its executions are linearizable [10]. The operation $R.\text{snapshot}()$ can be implemented with the well-known "double scan" technique (as used in [1]), where each process p_i is provided with an additional local sequence number sn_i, which it uses to identify all its write invocations (namely, when it invokes $R.\text{write}(x, v)$, p_i actually issues the following sequence of statements "$sn_i \leftarrow sn_i + 1$; $R[x] \leftarrow (v, id_i, sn_i)$". As no two processes have the same identity, each invocation of $R.\text{write}()$ is unambiguously identified.[2] To not overload the presentation, the sequence

---

[2] The proof of the operations $R.\text{write}()$, $R.\text{read}()$, and $R.\text{snapshot}()$ terminate and are linearizable is the same as the one in [1]. As far as the proof for $R.\text{snapshot}()$ is concerned, this comes from the observation that, in the algorithm described in [1], the order in which a process scans the array $R[1..m]$ is irrelevant. The important point is that, after it sequentially scanned twice the array $R[1..m]$, a process compares the corresponding entries of the two copies of $R[1..m]$ it has obtained. In our case, for any x, the first read of $R[x]$ and a second read of $R[x]$ by the same process are on the very same memory location, from which follows that the process correctly compares the corresponding entries of the two copies it has obtained to see if $R[1..m]$ changed between the two consecutive scans.

numbers associated with the write operations (and used in the "double scan" inside the snapshot operations) are left implicit in the rest of the paper[3].

RMW anonymous communication model: This is the RW model where, in addition to the operations $R.\text{read}(x)$ and $R.\text{write}(x, v)$, a process can also invoke an additional operation denoted $R.\text{compare\&swap}(x, -, -)$ defined in Section I-C. (The deadlock-free mutex algorithm described in Section V does not use the operation $R.\text{snapshot}()$.)

C. Symmetric Algorithm

The notion of a *symmetric algorithm* dates back to the eighties [6], [11]. Here, as in [21], a *symmetric algorithm* is an algorithm in which there is no way for two processes to distinguish one from the other.

This means that the process identities are defined by a specific data type, which allows to compare two identities for equality only. Hence process identities are not convertible to other data types, such as integers to know (the odd/even notion is meaningless for this data type). This type is similar to the Boolean data type in the sense that while Boolean values can be compared, there no odd/even notion for Boolean values).

Let us notice that the previous notion of symmetry can be seen as a weak notion of process anonymity in the sense it is not possible to define a ranking function on processes that could be exploited to solve mutual exclusion.

D. Initial Values

Moreover, in order for the initial values not to be used to destroy anonymity (which could favor a given process), all registers are initialized to the same value, namely the default value $\perp$.

E. Mutual Exclusion

Mutual exclusion is the oldest and one of the most important synchronization problems. Formalized by E. W. Dijkstra in the mid-sixties [4], it consists in building what is called a lock (or mutex) object, defined by two operations, denoted lock() and unlock(). (Recent textbooks including mutual exclusion and variants of it are [17], [20].)

The invocation of these operations by a process p_i always follows the following pattern: "lock(); *critical section*; unlock()", where "critical section" is any sequence of code. A process that is not in the critical section and has no pending lock() or unlock() invocation, is said to be in the *remainder section*. An infinite execution is *fair*, if every process that has a pending lock() or unlock() invocation, either finishes its operation or executes infinitely many steps.

---

[3]Defining each register as a record which has two fields (a value field and a sequence number field) with global (non-anonymous) names is done only for convenience. The two values in these fields can be encoded as a single value, removing the need for using more than one field.

The mutex object satisfying the deadlock-freedom progress condition is defined by the following two properties.

- Mutual exclusion: No two processes are simultaneously in their critical section.
- Deadlock-freedom: If a process p_i has a pending lock() or unlock() invocation and no process is in the critical section, then some process p_j (possibly $p_j \neq p_i$) eventually finishes its lock() or unlock() operation, provided the execution is fair.

III. ANONYMOUS RW MODEL: SYMMETRIC DEADLOCK-FREE MUTUAL EXCLUSION

This section presents Algorithm 1, that is a symmetric (with respect to equality) deadlock-free mutex algorithm suited to the RW memory-anonymous communication model. As indicated in the introduction, as this algorithm works for the necessary condition $(m > 1) \wedge (m \in M(n))$, its existence proves that this condition is also sufficient.

A. Representation of the Lock Object

Shared memory: Let m be such that $m > 1$ and $m \in M(n)$. The shared memory is composed of a memory-anonymous array $R[1..m]$, as defined in Section II-A.

For any $x \in \{1, \ldots, m\}$, the initial value of a register $R[x]$ is $\perp$. If $R[x] \neq \perp$, it contains the identity of the last process that wrote in this register. Given any execution and any of its configuration C, we use the following terminology (intuitively, a *configuration* is a consistent global state –local state of each process and value of each anonymous register– See [13] for a formal definition):

- "process p_i owns $R[x]$", if $R[x] = id_i$;
- "register $R[x]$ is owned" if $R[x] \neq \perp$;
- "R is full" if all its entries are owned;
- "R is empty" if none of its entries are owned.

Local memory: Each process p_i manages two local variables: an integer denoted cnt_i, and a local array denoted $view_i[1..m]$. The aim of $view_i[1..m]$ is to contain the value of R obtained by p_i from its last invocation of $R.\text{snapshot}()$. To prevent confusion, the shared array $R[1..m]$ is denoted with an uppercase letter, while the local variables are denoted with lowercase letters. As already indicated, the local sequence number associated with each write operation of a process p_i is left implicit.

B. Algorithm 1

Underlying principles: The core of Algorithm 1 consists in managing a competition among the processes until all the entries of $R[1..m]$ contain the same process identity, the corresponding process being the winner.

When a process invokes unlock(), or when it concludes while competing that it will not be the winner, it resets the entries of $R[1..m]$ containing its identity to $\perp$ (their initial value).
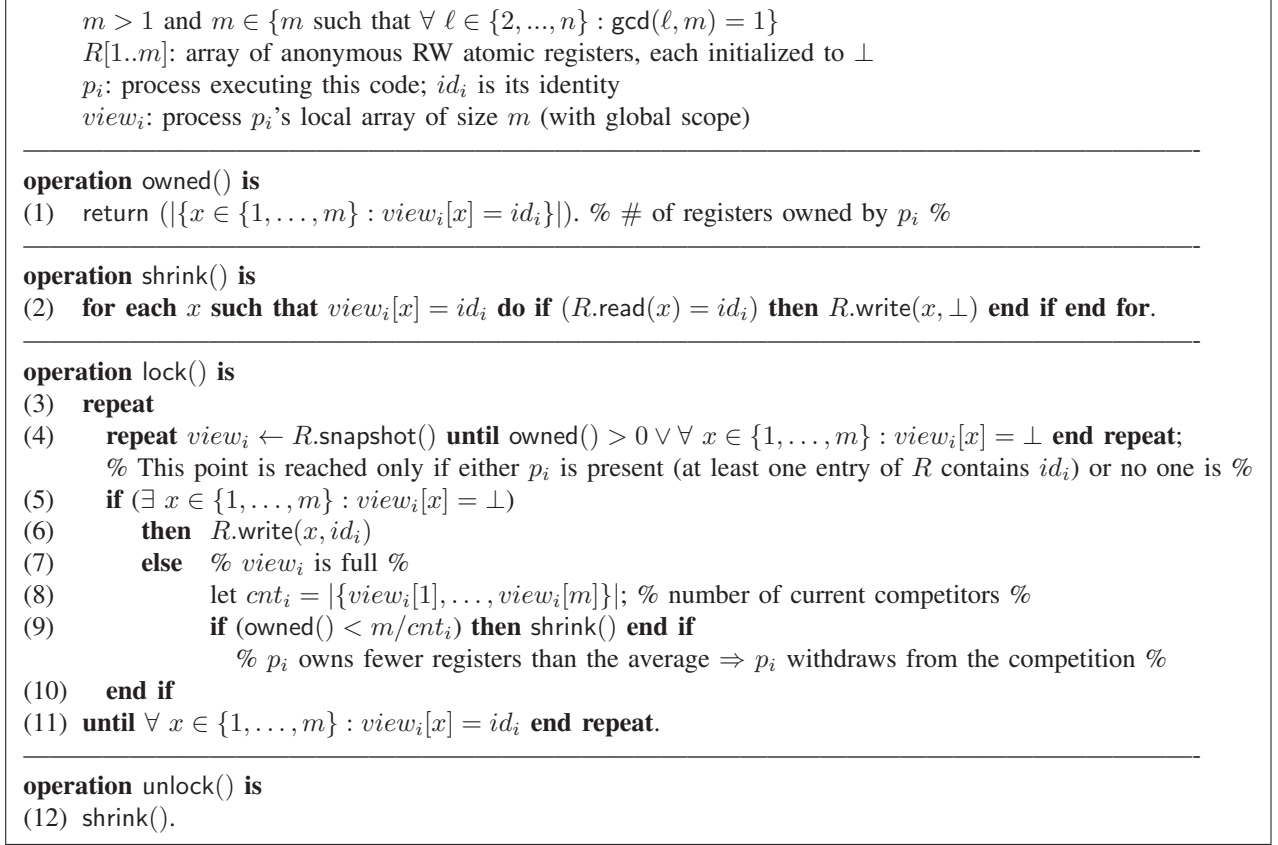
Hence, the core of the algorithm lies in the definition of predicates that direct a process to either withdraw from the competition or continue competing. To this end, a process p_i checks whether its local view $view_i$ (obtained from the invocation of $R.\mathsf{snapshot}()$) is full (line 5), and if so, whether p_i owns less than the average of all registers present in the competition (line 9).

Detailed description of the algorithm: Operation unlock() is a simple invocation of an internal operation called shrink() (line 12). With a shrink() invocation, a process p_i removes itself from the array by considering its latest view, $view_i$. More specifically, for each index $x \in \{1, \ldots m\}$ with $view_i[x] = id_i$, process p_i first reads $R[x]$, and if $R[x]$ still equals id_i, it writes $\perp$ into $R[x]$ (line 2).

The core of the algorithm is the code of the operation lock(). When a process p_i invokes this operation, it enters a "repeat-until" loop from which it can only exit when it obtains a snapshot of the anonymous shared memory $R[1..m]$, according to which p_i owns all entries (lines 3-11).

Hence, process p_i repeatedly invokes $R.\mathsf{snapshot}()$ (line 4) until it sees only the default value $\perp$ in the array $view_i$ obtained from $R.\mathsf{snapshot}()$ (which means that, from its local point of view, there is no competition), or it is present in this array (which means it is already competing). Then, when it stops looping, p_i checks whether $view_i$ is full (line 5) to know if it should continue writing (line 6) or if it should consider withdrawing from the competition (lines 7-9).

If $view_i$ is full, processes are engaged in a competition. If its identity appears in fewer than the average number of owned registers, process p_i withdraws from the competition by invoking the operation shrink() (lines 7-9), which suppresses its identity from the anonymous RW memory $R[1..m]$. After finishing its shrink() invocation, a process re-enters the repeat-until loop at line 4. The fact that $m \in M(n)$ guarantees that not all processes that appear in R when it is full, own the same number of registers, so at least one process will withdraw. If a process owns at least the average number of registers when its view is full, it re-enters the repeat-until loop and invokes the operation snapshot() again at line 4.

If $view_i$ is not full and p_i owns at least one register, it continues competing. To this end, before re-entering the repeat-until loop, p_i chooses an entry of $R[1..m]$ equal to $\perp$, and writes its identifier id_i in this register (lines 5-6).

To summarize, during a lock() operation, a process p_i

decides its future steps based on its latest view of the anonymous memory as follows:

1) If p_i owns all registers, it enters the critical section (line 11).
2) If p_i owns no register, and the view is not empty, then it waits (by repeatedly taking snapshots) until it obtained an empty view (line 4).
3) If the view is full, and cnt_i different processes own some registers, and p_i owns fewer than m/cnt_i registers, then it removes itself from all registers it owns by calling shrink() (line 9).
4) If the view is not full, there is at least one register that is not owned, and p_i writes its identity id_i into any not owned register (line 6).

IV. ANONYMOUS RW MODEL: PROOF OF ALGORITHM 1 AND TIGHT SPACE LOWER BOUND

A. RW Memory-Anonymous Model: Proof of Algorithm 1

Let us remember that the anonymous RW array $R[1..m]$ is the only object that the processes can use to communicate. The notions of "time", "first" and "last" used in the proofs are well-defined, as all write() and read() operations are atomic. As stated in Section II-A, the proof assumes that operation snapshot() (which can be implemented from atomic read/write operations) is linearizable and satisfies the progress condition 1. The proof assumes $n \geq 2$, as otherwise mutual exclusion is trivial. Moreover, let us remember that m is assumed to be greater than 1 and belong to the set $M(n) = \{m : \forall \ell : 1 < \ell \leq n : \ \gcd(\ell, m) = 1\}$, from which it follows that $m > n$.

Let E be an arbitrary infinite execution E, $L(E)$ an execution where all snapshot() operations occur atomically at their linearization points (i.e., $L(E)$ is a linearization of all operations on R in E).

Theorem 1. *Algorithm* 1 *satisfies mutual exclusion.*

Proof Consider history $L(E)$. Let us suppose by contradiction that two processes are inside their critical section at the same time, and assume that p_i is the first of them to take its last snapshot before entering its critical section. More precisely, suppose process p_i's lock() invocation terminates (and thus p_i enters the critical section) following some iteration of the outer repeat-until loop in lock(). Then due to the predicate of line 11, p_i owns all registers of R at the point of p_i's snapshot() (line 4) in its last iteration. Therefore, in the same iteration the predicate of line 5 and the predicate of lines 9 are false, and the predicate of lines 11 is true. Hence, the snapshot() in line 4 at the beginning of the iteration is p_i's last access to the RW anonymous memory before its lock() operation terminates. We therefore say a *process enters the critical section* at the point when it is taking a snapshot in line 4 while owning all registers.

Now suppose p_i enters the critical section at some point t. Let t' be the point when p_i executes its first shared memory

operation in its subsequent unlock() invocation, if there is such an invocation, and otherwise $t' = \infty$. Let us consider the following claim:

Claim 1. *Throughout* $[t, t']$, *the value of the anonymous memory returned by all the invocations of* snapshot() *contains the identity of* p_i.

It follows from this claim that at no point in $[t, t']$ a process other than p_i can observe itself as owning all registers. Also, as assumed at the beginning of the proof, process p_i is the *first* to take its last snapshot before entering its critical section. Thus no other process, except p_i, can be in the critical section throughout $[t, t']$, which contradicts the assumption that p_i is not alone in the critical section.

Proof of the claim. For the purpose of a contradiction, let us assume that Claim 1 is not true. Because all m registers are owned by p_i at time t and $m > n$, by the pigeonhole principle, at least one process has issued more than one write that changed the value of a register from the identity of p_i to another value. Let p_j be the first process to do so. Hence, process p_j took a snapshot at some point $T_s \in [t, t']$ in line 4 at the beginning of the iteration of the outer repeat-until loop in which it executes its second R.write() in line 5, that changes the value of a register from the identity of p_i to another value.

Process p_i is the only process that can write its own identity, it owns all the registers at time t, and it does not execute any write operation in $[t, t']$. Then, in the snapshot taken at $T_s \in [t, t']$ by p_j, the second register overwritten by p_j contains p_i's identity, and is not chosen at line 5. A contradiction which completes the proof of the claim and the theorem. $\square_{Theorem\ 1}$

Theorem 2. *Algorithm* 1 *is deadlock-free.*

The remainder of this section is devoted to the proof of this theorem. For the purpose of contradiction let us assume that E is an infinite fair execution, and that after some point t^* no invocation of lock() or unlock() terminate, even though at least one process has a pending lock() or unlock() operation and no process is in the critical section. Since unlock() is wait-free [9] (see also the Claim 2 below) we may assume w.l.o.g. that no invocation of unlock() is pending at any point after t^*.

Definitions.
- A process is *shrinking* if it is poised to read or write in a shrink() operation.
- A process is *large* if it owns at least 2 registers.

Claim 2. *In any execution, each invocation of* shrink() *by a process* p_i *terminates within a finite number of* p_i's *steps, and when it does, process* p_i *owns no register.*

Proof Process p_i executes at most m iterations of the for-loop in shrink(), and in each iteration it executes the wait-free operations write() and read(), so shrink() terminates after a finite number of p_i's steps. Before calling shrink(), process p_i calls snapshot() to obtain $view_i$ (line 6), and it does not write to the RW anonymous memory $R[1..m]$ between that snapshot() and its subsequent call of shrink(). In the for-loop in the operation shrink(), process p_i writes $\perp$ into all registers $R[x]$, $x \in \{1, \ldots, m\}$, such that $view_i[x] = id_i$. Since no other process writes the value id_i to any register $R[x]$, no register contains id_i anymore when p_i terminates its shrink() operation. $\quad \square_{Claim\ 2}$

Claim 3. *If a process owns a register, then it is not in the remainder section (i.e., it has a pending* lock() *or* unlock() *invocation, or it is in the critical section).*

Proof A process p_i can only begin to own a register $R[x]$, $x \in \{1, \ldots, m\}$, by writing id_i into $R[x]$, that can only happen in line 12 of the operation lock(). When p_i enters the remainder section, it has not written to any register of R since its latest shrink() invocation in the operation unlock() terminated, so the statement follows from the Claim 2. $\quad \square_{Claim\ 3}$

In the following, for any given time $t \geq t^*$, we say that p_i is *competing* if p_i has a pending lock() operation at t and the last snapshot taken by p_i before t satisfies the condition at line 4 (i.e. p_i is not stuck in the inner loop).

Claim 4. *At any point $t \geq t^*$, there is a competing process p_i whose last* snapshot() *invocation does not cause it to invoke* shrink() *at line 9.*

Proof If at least one competing process p_i obtains a view that is not full, the condition at line 5 is satisfied, and thus this view does not cause p_i to invoke shrink(). We can then consider that all competing processes have obtained a full view in their last snapshot.

Let p_i be the process that owns the most registers in the last snapshot taken before t (if more than one process satisfy this condition, p_i can be chosen arbitrarily among them). If p_i took this snapshot, it wouldn't cause it to invoke shrink() (p_i owns more than the average, condition at line 9). Let us then consider that p_i didn't take this last snapshot before t, but took one previously at time $t' < t$. Process p_i is the only one which can write its own identity, and its last view was full, causing it not to write (condition at line 4). At time t', p_i then owns at least as many registers as in the last snapshot taken before t. Furthermore, any competing process in the last snapshot taken before t is also competing at time t' (otherwise it would be stuck in the inner loop at line 4). Thus, the view taken by p_i at time t' does not satisfy the condition at line 9, and does not cause p_i to invoke shrink(). $\quad \square_{Claim\ 4}$

Claim 5. *At any point $t \geq t^*$, if there is more than one competing process, at least one of them will invoke* shrink().

Proof Suppose not. Note that the only point at which a process can write $\perp$ is during the shrink() operation. If at least one competing process obtains a view that is not full, it will invoke R.write(). This will happen again until no register has the value $\perp$ and all competing processes obtain full views in their last snapshot, preventing them from writing. We can then consider w.l.o.g. that, at time t, all competing processes have stopped writing and have obtained the same view. Let cnt be the number of these competing processes.

Because $1 < cnt \leq n$ and $\forall \ell : 1 < \ell \leq n : \gcd(\ell, m) = 1$, at least one competing process owns less registers than m/cnt, causing it to call shrink(); a contradiction which proves the claim. $\quad \square_{Claim\ 5}$

Proof of Theorem 2.
By Claim 4, at any point $t \geq t^*$, there is a competing process whose last snapshot() invocation does not cause it to invoke shrink(). By Claim 2, any shrink() operation terminates, and causes the invoking process to be stuck in the inner loop at line 4, causing it to stop competing after its next snapshot() invocation. This implies that at least one competing process never calls shrink() after point t^*.

By assumption, no process is in the critical section, and no unlock() operation is pending. By Claim 3, if a process owns a register, then it is not in the remainder section. The only processes that own registers are then the ones that are competing.

By Claim 5, if there is more than one competing process, at least one of them invokes shrink(), causing it to stop competing. There is then eventually a single competing process that owns all the registers, a contradiction with the original assumption that after some point t^*, no invocation of lock() or unlock() terminates, even though at least one process has a pending lock() or unlock() operation and no process is in the critical section. $\quad \square_{Theorem\ 2}$

B. Anonymous RW Model: Tight Space Bound

Given $M(n) = \{m : \forall \ell : 1 < \ell \leq n : \gcd(\ell, m) = 1\}$, it is shown in [21] that $m \in M(n)$ is a necessary condition for any algorithm solving symmetric deadlock-free mutex in an anonymous memory composed of m read/write registers. As already indicated, as $m \geq n$ is a necessary condition for any algorithm solving deadlock-free mutex in a non-anonymous system, it remains necessary in a read/write anonymous system. This translates as follows: $m \in M(n) \setminus \{1\}$ is a necessary condition for deadlock-free mutex in an anonymous memory composed of m read/write registers.

As Algorithm 1 solves deadlock-free mutex under this condition, it follows that $m \in M(n) \setminus \{1\}$ is a necessary and sufficient condition.

V. Anonymous RMW Model: Symmetric Deadlock-Free Mutual Exclusion

This section presents an algorithm implementing a deadlock-free mutex lock object in an n-process read/modify/write (RMW) memory-anonymous system. As the previous algorithm, the algorithm presented below is particularly simple.

A. Representation of the Lock Object

The shared anonymous memory is made up of m RMW atomic registers, denoted $R[1..m]$ where $m \in \{1\} \cup \{m : \forall \ell : 1 < \ell \leq n : \gcd(\ell, m) = 1\}$ (let us notice that this set includes the value 1).

In Algorithm 2, a process uses three local variables, denoted $most\_present_i$, $owned_i$, and $view_i$ (which has the same meaning as in Algorithm 1).

B. Algorithm 2

When a process p_i invokes lock(), it enters a repeat loop from which it will exit when it will obtain a view $view_i[1..m]$ in which its own identity appears in a majority of registers (line 12).

Process p_i first invokes the RMW operation compare&swap() on all registers in order to write its identity in all the registers whose current value is the default value $\perp$ (line 2). Then, it reads (asynchronously) the whole anonymous memory and saves it in $view_i[1..m]$ (line 3). From this non-atomic view of the shared memory, p_i computes the occurrence number of the most present value ($most\_present_i$, line 4) and the occurrence number of its own identity ($owned_i$, line 5).

- If $owned_i \geq most\_present_i$, p_i proceeds to the next iteration of the repeat-until loop if $owned_i \leq m/2$, and enters the critical section if $owned_i > m/2$.
- If $owned_i < most\_present_i$, p_i resigns from the competition. To this end, it first writes $\perp$ in all entries that, from its local point of view, contain its identity (line 7), and then waits until it sees that all the anonymous registers contain the default value $\perp$ (lines 8-10).

When a process p_i invokes unlock(), it simply resets to $\perp$ all the registers that contain its identity id_i (line 13).

VI. Anonymous RMW Model: Proof of Algorithm 2 and Tight Space Lower Bound

A. Proof of Algorithm 2

Theorem 3. *Algorithm* 2 *satisfies mutual exclusion.*

Proof Assume that a process p_i is in its critical section, while some other process, say process p_j, is executing the operation lock(). Before p_i entered its critical section the exit predicate of line 12, namely, $owned_i > m/2$ must be evaluated to true. This means that, before p_i entered its critical section, it succeeded to change more than $m/2$ RMW anonymous registers from $\perp$ to its identifier id_i. As

long as process p_i does not set these RMW registers back to $\perp$, process p_j cannot succeed in changing more than $m/2$ registers from $\perp$ to id_j. Thus, process p_j will not be able to enter its critical section while p_i is in its critical section.
$\square_{Theorem\ 3}$

Theorem 4. *Algorithm* 2 *is deadlock-free.*

Proof We show that if a process is trying to enter its critical section, then some process eventually enters its critical section.

In the first for loop (line 2) each process scans the m RMW anonymous registers trying to set those that are $\perp$ to its identifier. If the process is running alone, it will clearly succeed to set them all to its identifier and will enter its critical section.

When there is contention (i.e., several processes are in their entry codes) since $\forall x \in \{1, ..., n\} : m$ and x are relatively prime, at least one of the processes p_k must find that less than $most\_present_k$ of the RMW registers are set to its identifier. It follows from lines 6-7 that p_k gives up the competition, and waits in the inner repeat loop (lines 8-10). This enables at least one other process p_j, for which $most\_present_j$ of the RMW registers are set to its identifier, to proceed. Repeating this argument, eventually one of the processes will find that its identifier appears in more than $m/2$ of the RMW registers and will enter its critical section.

Finally, as in its exit code (line 13), a process sets to $\perp$ all the registers containing its identifier. This enables a possibly waiting process to continue. Thus, it is not possible for all the processes to simultaneously remain forever in the operation lock().
$\square_{Theorem\ 4}$

B. Anonymous RMW Model: Tight Space Lower Bound

Theorem 5. *There is an n-process symmetric deadlock-free mutual exclusion algorithm using $m \geq 1$ anonymous RMW registers only if $m \in M(n) = \{m : \forall \ell : 1 < \ell \leq n : \gcd(\ell, m) = 1\}$.*

Proof Let us assume to the contrary, namely, there is a symmetric deadlock-free mutual exclusion algorithm for n processes using $m \geq 1$ anonymous RMW registers such that for some positive integer $1 < \ell \leq n$, m and ℓ are not relatively prime. This means that there is a number $1 < \ell \leq m$ such that ℓ divides m.

Let us arrange the m RMW registers on a ring with m nodes where each register is placed on a different node. Then, let us pick ℓ processes. For simplicity let us call these processes $p_0, ..., p_{\ell-1}$. To each one of the ℓ processes, we assign an initial RMW register (namely, the first register that the process accesses) such that for every two processes p_i and $p_{i+1 \ (mod\ \ell)}$, the distance between their initial registers is exactly m/ℓ when walking on the ring in a clockwise direction. Here we use the assumption that ℓ divides m.

```
m ∈ {m such that ∀ ℓ ∈ {2, ..., n} : gcd(ℓ, m) = 1}
R[1..m]: array of anonymous RMW atomic registers, each initialized to ⊥
pᵢ: process executing this code; idᵢ is its identity
viewᵢ: process pᵢ's local array of size m (with global scope)
───────────────────────────────────────────────────────────────────────
operation owned() is return (|{x ∈ {1, ..., m} : viewᵢ[x] = idᵢ}|). % # of registers owned by pᵢ %
───────────────────────────────────────────────────────────────────────
operation lock() is
(1)   repeat
(2)      for each x ∈ {1, ..., m} do R.compare&swap(x, ⊥, idᵢ) end for;
(3)      for each x ∈ {1, ..., m} do viewᵢ[x] ← R.read(x) end for;
(4)      most_presentᵢ ← maximum number of times the same non-⊥ value appears in viewᵢ;
(5)      ownedᵢ ← owned();
(6)      if ownedᵢ < most_presentᵢ then
(7)         for each x ∈ {1, ..., m} do if (viewᵢ[x] = idᵢ) then R.write(x, ⊥) end if end for;
(8)         repeat
(9)            for each x ∈ {1, ..., m} do viewᵢ[x] ← R.read(x) end for
(10)        until ∀ x ∈ {1, ..., m} : viewᵢ[x] = ⊥ end repeat
(11)     end if
(12)  until ownedᵢ > m/2 end repeat.
───────────────────────────────────────────────────────────────────────
operation unlock() is
(13)  for each x ∈ {1, ..., m} do R.compare&swap(x, idᵢ, ⊥) end for.
```

```
m ∈ {m such that ∀ ℓ ∈ {2,...,n} : gcd(ℓ,m) = 1}
R[1..m]: array of anonymous RMW atomic registers, each initialized to ⊥
p_i: process executing this code; id_i is its identity
view_i: process p_i's local array of size m (with global scope)
────────────────────────────────────────────────────────────────────────
operation owned() is return (|{x ∈ {1,...,m} : view_i[x] = id_i}|). % # of registers owned by p_i %
────────────────────────────────────────────────────────────────────────
operation lock() is
(1)    repeat
(2)       for each x ∈ {1,...,m} do R.compare&swap(x, ⊥, id_i) end for;
(3)       for each x ∈ {1,...,m} do view_i[x] ← R.read(x) end for;
(4)       most_present_i ← maximum number of times the same non-⊥ value appears in view_i;
(5)       owned_i ← owned();
(6)       if owned_i < most_present_i then
(7)          for each x ∈ {1,...,m} do if (view_i[x] = id_i) then R.write(x, ⊥) end if end for;
(8)          repeat
(9)             for each x ∈ {1,...,m} do view_i[x] ← R.read(x) end for
(10)         until ∀ x ∈ {1,...,m} : view_i[x] = ⊥ end repeat
(11)      end if
(12)   until owned_i > m/2 end repeat.
────────────────────────────────────────────────────────────────────────
operation unlock() is
(13)   for each x ∈ {1,...,m} do R.compare&swap(x, id_i, ⊥) end for.
```

Figure 2. (Algorithm 2) RMW mem.-anonymous deadlock-free mutex (n-process system, $n \geq 2$, code for p_i)

The lack of global names, allows us to assign for each process an initial RMW register and an ordering of the registers which determines how the process scans the registers.

An execution in which the ℓ processes are running in *lock steps*, is an execution where we let each process take one step (in the order $p_0, ..., p_{\ell-1}$), and then let each process take another step, and so on. For process p_i and integer k, let $order(p_i, k)$ denotes the kth new register that p_i accesses during an execution where the ℓ processes are running in lock steps, and assume that we arrange that $order(p_i, k)$ is the register whose its distance from p_i's initial registers is exactly $(k-1)$, when walking on the ring in a clockwise direction.

We notice that $order(p_i, 1)$ is p_i's initial register, $order(p_i, 2)$ is the next new register that p_i accesses and so on. That is, p_i does not access $order(p_i, k+1)$ before accessing $order(p_i, k)$ at least once, but for every $j \leq k$, p_i may access $order(p_i, j)$ several times before accessing $order(p_i, k+1)$ for the first time.

With this arrangement of RMW registers, we run the ℓ processes in lock steps. Since only comparisons for equality are allowed, and all registers are initialized to a the same value –which (to preserve anonymity) is not a process identity– processes that take the same number of steps will be at the same state, and thus it is not possible to break symmetry. It follows that either all the processes will enter their critical sections at the same time, violating mutual exclusion, or no process will ever enter its critical section, violating deadlock-freedom. A contradiction. □$_{Theorem\ 5}$

Registers	RW anonymous	RMW anonymous
Sufficient condition	This paper (Algorithm 1)	This paper (Algorithm 2)
Necessary condition	[21][4]	This paper (Theorem 5)

Table II
A GLOBAL PICTURE FOR n-PROCESS ANONYMOUS MUTEX ($n \geq 2$)

VII. CONCLUSION

"Anonymous shared memory" means there is no a priori agreement among the processes on the names of the shared registers. Moreover, "symmetric algorithm" means that the process identities define a specific data type with no internal structure (such as a total order) and no relation with other data type (hence an identity cannot be compared with an integer). Identities can only be read, written, and compared with equality.

Considering two types of anonymous registers (the atomic read/write (RW) registers and the atomic read/modify/write

─────────

[4]Notice that the lower bound for the RW model from [21], follows immediately from our stronger lower bound for the RMW model present in this paper.

(RMW) registers) this paper presented several results on symmetric mutual exclusion algorithms, summarized in Table II. These algorithms differ in their design principles and their costs (measured as the number of registers which must contain the identity of a process to allow it to enter the critical section).

The symmetric deadlock-free mutex algorithm built on top of an anonymous memory made up of m atomic RW registers works for $m \in M(n) \setminus \{1\}$, where $M(n) = \{m : \forall \ell : 1 < \ell \le n : \gcd(\ell, m) = 1\}$, while the algorithm for m atomic RMW registers works for $m \in M(n)$. The necessity of the first condition was proved in [21], while the necessity of the second condition was proved in this paper. The existence of the algorithms presented in the paper proves these conditions are also sufficient.

Let us remark that a system composed of a single anonymous register is no really anonymous. Hence, if we eliminate the "pathological" case $m = 1$, the condition $m \in M(n)$ is a necessary and sufficient for symmetric deadlock-free mutex in both the read/write and the read/modify/write anonymous register models. This shows a fundamental computability difference separating the "memory anonymity" adversary (which operates before the execution and is consequently *static*) and the "process crash" adversary (which operates during the execution and is consequently *dynamic*), for which read/write and read/modify/write registers (provided with an operation such as compare&swap) are located at the two extremes of the synchronization power hierarchy as defined in [9]. (Let us remind that mutex can be solved neither in the read/write nor in the read/modify/write non-anonymous register models in the presence of process crashes.) Last but not least, a noteworthy property of the two algorithms that have been presented lies in their simplicity.

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

[2] Anderson J., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

[3] Burns J. E. and Lynch N. A., Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171-184 (1993)

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

[5] Fischer M. J., Lynch N. A., and Paterson M. S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[6] Garg V.K. and Ghosh J., Symmetry in spite of hierarchy. *Proc. 10th Int'l Conference on Distributed Computing Systems (ICDCS'90)*, IEEE Computer Press, pp. 4-11 (1990)

[7] Giakkoupis G., Helmi M., Higham L., and Woelfel P., An O(sqrt n) space bound for obstruction-free leader election. *Proc. 27th Int'l Symposium Distributed Computing (DISC'13)*, Springer, LNCS 8205, pp. 46-60 (2013)

[8] Giakkoupis G., and Woelfel P., Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC Model. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 221-229 (2017)

[9] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149 (1991)

[10] Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*,12(3):463-492, (1990)

[11] Johnson R.E., and Schneider F.B., Symmetry and similarity in distributed systems. *Proc.4th ACM Symposium on Principles of Distributed Computing (PODC'85)*, pp. 13-22, ACM Press (1985)

[12] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[13] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press (1987)

[14] Navlakha S. and Bar-Joseph Z., Algorithms in nature: the convergence of systems biology and computational thinking. *Molecular systems biology*, 7(546):1-11 (2011)

[15] Navlakha S. and Bar-Joseph Z., Distributed information processing in biological and computational systems. *Communications of the ACM*, 58(1):94-102 (2015)

[16] Rabin M., The choice coordination problem. *Acta Informatica*, 17(2):121-134 (1982)

[17] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[18] Raynal M. and Cao J., Anonymity in distributed read/write systems: an introductory survey. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS, 17 pages (2018)

[19] Rashid S., Taubenfeld G., and Bar-Joseph Z., Genome wide epigenetic modifications as a shared memory consensus. *6th Workshop on Biological Distributed Algorithms (BDA'18)*, London (2018)

[20] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)

[21] Taubenfeld G., Coordination without prior agreement. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 325-334 (2017)