Memory-Anonymous Starvation-Free Mutual Exclusion: Possibility and Impossibility Results

Gadi Taubenfeld
Reichman University, Herzliya, Israel

Abstract

In an anonymous shared memory system, all inter-process communications are via shared objects; however, unlike in standard systems, there is no a priori agreement between processes on the names of shared objects [14, 15]. Furthermore, the algorithms are required to be symmetric; that is, the processes should execute precisely the same code, and the only way to distinguish processes is by comparing identifiers for equality. For such a system, read/write registers are called anonymous registers. It is known that symmetric deadlock-free mutual exclusion is solvable for any finite number of processes using anonymous registers [1]. The main question left open in [14, 15] is the existence of starvation-free mutual exclusion algorithms for two or more processes. We resolve this open question for memoryless algorithms, in which a process that tries to enter its critical section does not use any information about its previous attempts. Almost all known mutual exclusion algorithms are memoryless. We show that,

1. There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers if and only if m is odd.
2. There is no symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using (any number of) anonymous registers.

Our impossibility result is the only example of a system with fault-free processes, where global progress (i.e., deadlock-freedom) can be ensured, while individual progress to each process (i.e., starvation-freedom) cannot. It complements a known result for systems with failure-prone processes, that there are objects with lock-free implementations but without wait-free implementations [2, 5].

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Shared memory algorithms; Theory of computation → Distributed algorithms

Keywords and phrases anonymous shared memory, memory-anonymous algorithms, anonymous registers, starvation-free mutual exclusion

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.33

1 Introduction

1.1 Anonymous shared memory

A central issue in distributed systems is coordinating the actions of asynchronous processes. In the context where processes communicate via reading and writing from shared memory, in almost all published concurrent algorithms, it is assumed that the shared memory locations have global names, which are a priori known to all the participating processes. The intriguing question of what and how coordination can be achieved without relying on such lower-level agreement about the names of the memory locations was introduced and studied in [14, 15].

We assume that all inter-process communications are via shared read/write registers which are initially in a known state. However, unlike in the standard model, from the point of view of the processes, the registers do not have global names. Such registers are called *anonymous* registers. Algorithms correct for a model where the registers are anonymous are called *memory-anonymous* algorithms.

There are fundamental differences between the standard shared memory model and the strictly weaker anonymous shared memory model [15]. Besides enabling us to understand better the intrinsic limits for coordinating the actions of asynchronous processes, the anonymous

37th International Symposium on Distributed Computing (DISC 2023).
Editor: Rotem Oshman; Article No. 33; pp. 33:1–33:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

shared memory model with symmetric processes has been shown to be useful in modeling biologically inspired distributed computing methods, especially those based on ideas from molecular biology [9]. The main question left open in [14, 15] is regarding the existence of symmetric starvation-free mutual exclusion algorithms for two or more processes when using anonymous registers. In this article, we resolve this open question for memoryless algorithms.

1.2 Mutual exclusion, symmetric algorithms, memoryless algorithms

The mutual exclusion problem. The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among several competing processes [4]. It is assumed that each process executes a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical, and exit. The exit section is required to be wait-free – its execution must always terminate. It is assumed that processes do not fail and that a process always leaves its critical section. The *mutual exclusion problem* is writing the code for the entry and exit sections to satisfy the following two basic requirements.

- *Deadlock-freedom:* If a process tries to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- *Mutual exclusion:* No two processes are in their critical sections simultaneously.

The satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm. For an algorithm to be fair, the satisfaction of the following stronger progress condition is required.

- *Starvation-freedom:* If a process is trying to enter its critical section, then this process eventually enters its critical section.

Symmetric algorithms. A symmetric algorithm is an algorithm in which the processes execute exactly the same code, and the only way to distinguish processes is by comparing identifiers for equality. A process can determine if two identifiers are the same, but nothing else can be determined when they are different. Identifiers can be written, read, and compared, but there is no way of looking inside any identifier. Thus, for example, knowing whether an identifier is odd or even is impossible.

Furthermore, (1) a process can only compare its identifier with another and cannot compare it with a constant value, and (2) the local variables of the different processes have the same names, and (local variables with the same names) are initialized to the same values. Otherwise, it would be possible to distinguish different processes. In symmetric algorithms, as defined above, we say that the processes are symmetric.

As symmetric algorithms do not depend on an order relation between process identities, they require fewer assumptions and are consequently more general than non-symmetric algorithms. The symmetry constraint on process identities can be seen as the “last step” before process anonymity [10, 11]. Symmetric algorithms with non-anonymous memory have been investigated for years [12]. Following [14], we consider a model in which the memory is anonymous, and the processes are symmetric.

Memoryless algorithms. A memoryless mutual exclusion algorithm is an algorithm in which when all the processes are in their remainder section, the values of all the registers (local and shared) are the same as their initial values. This means that a process that tries to enter its critical section does not use any information about its previous attempts (like the fact that it has entered its critical section five times so far). Put another way, in a memoryless algorithm, processes have only a single remainder state and hence cannot retain any memory of prior executions of the algorithm.

All known mutual exclusion algorithms which use anonymous registers and almost all known mutual exclusion algorithms that use non-anonymous registers are memoryless. For example, Lamport's Bakery algorithm is memoryless [8]. Memoryless mutual exclusion algorithms are usually simpler to understand than those that are not memoryless. Furthermore, memoryless algorithms can better handle system-wide failures (i.e., all processes crash simultaneously), as upon recovery, the system can be initialized to the (single) initial state, not affecting processes that had not participated in the algorithm during the failure. When using non-anonymous registers, $2n-1$ registers are necessary and sufficient for designing a symmetric memoryless starvation-free mutual exclusion algorithm for n processes [12].

1.3 Contributions: individual progress vs. global progress

It is known that symmetric memoryless deadlock-free mutual exclusion is solvable for any finite number of processes using anonymous registers [1]. The main question left open in [14, 15] is the existence of symmetric starvation-free mutual exclusion algorithms for two or more processes. We resolve this open question for memoryless algorithms by proving the following possibility and impossibility results,

1. There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers if and only if m is odd.
2. There is no symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using (any number of) anonymous registers.

Our possibility result shows that (1) there is no separation between deadlock-freedom and starvation-freedom for two processes, and (2) there is a separation between deadlock-freedom and starvation-freedom for three or more processes. These results enable us to understand better the intrinsic limits for achieving fairness between asynchronous processes.

Interestingly, as a byproduct of the proof of our impossibility result, we get a general time complexity lower bound for every symmetric *deadlock-free* mutual exclusion algorithm for $n \geq 2$ processes using m anonymous registers. Namely, a process must incur $\lceil m/2 \rceil$ remote memory references (RMR) to enter and exit its critical section once. The RMR complexity of our starvation-free algorithm is $O(m)$.

Two main progress conditions have been studied for asynchronous shared-memory systems with failure-prone processes. The first, wait-freedom, ensures individual progress to each process, i.e., its operations complete as long as it takes an infinite number of steps [6]. The second, lock-freedom, requires only global progress; namely, if a process takes an infinite number of steps, then some (possibly other) processes complete their operations [7]. Wait-freedom corresponds to starvation-freedom in fault-free systems, while lock-freedom corresponds to deadlock-freedom.

It was shown in [5] that there is an object for which there is a lock-free implementation for two processes using only non-anonymous atomic registers, and there is no wait-free algorithm in the same setting. A more general result was presented in [2], showing that such a separation exists (1) also for more than two processes and (2) when primitives stronger than atomic registers are used. Our result, which shows a separation between deadlock-freedom and starvation-freedom for three or more processes, complements these results. The results together show that in shared memory systems where either failures are possible or process symmetry and memory anonymity are assumed, it is not always possible to ensure individual progress in situations where global progress is possible. Thus, achieving various levels of fairness depends on the underlying system assumptions.

2 Preliminaries

Processes. Our model of computation consists of a fully asynchronous collection of n deterministic processes that communicate via m anonymous registers. Asynchrony means that there is no assumption on the relative speeds of the processes. Each process has a unique identifier, which is a positive integer. Since we want to make as few assumptions as possible, it is *not* assumed that the identifiers of the n processes are taken from the set $\{1, ..., n\}$. Thus, a process does not a priori know the identifiers of the other processes. The processes do know the values of n and m. We assume that processes do not fail. As always assumed when solving the mutual exclusion problem, participation is not required – a process may stay in its remainder section and never move to its entry section.

Memory. The shared memory consists of m *anonymous* shared registers. For m anonymous registers, $r_1, ..., r_m$, the adversary can fix, for each process p, a permutation $\pi_p : \{r_1, ..., r_m\} \to \{r_1, ..., r_m\}$ of the registers such that, for process p, the j'th anonymous register is $\pi_p(r_j)$. In particular, when process p accesses its j'th anonymous register, it accesses $\pi_p(r_j)$. Algorithms designed for such a system must be correct regardless of the permutations chosen by the adversary. The permutation fixed for process p is called the naming assignment of p.

All the anonymous registers are assumed to be initialized to the same value. Otherwise, thanks to their different initial values, it would be possible to distinguish different registers, and consequently, the registers would no longer be fully anonymous.

With an *atomic* register, it is assumed that operations on the register occur in some definite order. That is, each operation is an indivisible action. In the sequel, by *registers*, we mean anonymous atomic read/write registers. A read/write register is a shared register that supports (atomic) read and write operations. The fact that anonymous registers do not have global names implies that only multi-writer multi-reader anonymous registers are possible. Such registers can both be written and read by all the processes.

Known results. In [14], it has been proven that a necessary and sufficient condition for the design of a symmetric deadlock-free mutual exclusion algorithm for two processes using anonymous registers is that the number of registers is odd.

▶ **Theorem 1** ([14]). *There is a symmetric deadlock-free mutual exclusion algorithm for two processes using $m \geq 2$ anonymous registers if and only if m is odd.*

We will use this result for two processes later in the paper. The *only if* part of the above theorem is a special case of the following more general result from [14] (Theorem 3.4): There is a symmetric deadlock-free mutual exclusion algorithm for $n \geq 2$ processes using $m \geq 2$ anonymous registers only if for every positive integer $1 < \ell \leq n$, m and ℓ are relatively prime. An optimal symmetric deadlock-free mutual exclusion algorithm using anonymous registers that matches the above general space bound for $n \geq 2$ processes was presented in [1].

3 A starvation-free mutual exclusion algorithm for two processes

We show that, for two processes, it is possible to design a starvation-free mutual exclusion algorithm. In the next section, we prove this is impossible for three or more processes.

▶ **Theorem 2.** *There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers if and only if m is odd.*

The *only if* direction follows from Theorem 1 (proven in [14]), where it has been proven that (when using anonymous registers) any symmetric deadlock-free mutual exclusion algorithm for two processes must use an odd number of anonymous registers. To prove the *if* direction, we present in Algorithm 1 a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using m anonymous registers, where m is an odd number greater than or equal to 7. The question of whether a symmetric memoryless starvation-free mutual exclusion algorithm exists for two processes using 3 or 5 anonymous registers is open.

As the m registers do not have global names, each process independently numbers them. We use the notation $p.i[j]$ to denote the j^{th} register according to process i's numbering, for $1 \leq j \leq m$. Recall that a process's identifier is a positive integer.

3.1 An informal description of the algorithm

A shared register is *free* when its value is 0. Initially, all the registers are free. A register is owned by process i, when its value is i. There are two ways for a process to get permission to enter its CS (i.e., Critical Section). The first way is when a process owns $m - 2$ registers. Initially, a process tries to own $m - 2$ registers by writing its identifier into free registers. If the process succeeds, it may enter its CS, and when done, it releases (i.e., sets to 0) all the registers it owns. By design, a process will never own more than $m - 2$ registers.

When there is contention, each process first tries to own as many registers as possible, but no more than $m - 2$ registers. Thus, each process will always succeed in owning at least one register (and not at least two registers, as explained in the sequel). After it attempts to own $m - 2$ registers, if a process notices that it owns less than $\lceil m/2 \rceil$ registers, it becomes a loser. A loser acts as follows: if it owns more than two registers, it releases all its owned registers except two. Otherwise, when a loser owns one or two registers, it releases no registers. Then, the loser writes "waiting" into the (one or two) registers it owns and waits. Since the waiting process owns at most two registers, the other process, the winner, keeps on trying to own more registers until it eventually succeeds in owning $m - 2$ registers and gets permission to enter its CS.

When a winning process exits its CS, the winner releases all the $m - 2$ registers it owns, which, as explained below, will prevent it from entering its CS again before a waiting process gets a chance to enter its CS. This guarantees that starvation-freedom is satisfied. To guarantee that one process will not enter its CS twice while the other process is waiting, when a process starts its entry code, it repeatedly scans the registers until none of them has the value "waiting," and only then may it proceed.

The second way a process can enter its CS is by waiting first. A waiting process owns one or two registers with the value "waiting." The waiting process waits until all the registers it does not own are released (i.e., have the value 0). Once this happens, the waiting process may immediately enter its CS. That is, it need not own additional registers. Upon exiting its CS, the (previously waiting) process releases the (one or two) registers it owns.

There is one very delicate possible race condition that should be avoided. Assume there is contention; process i writes its identifier into $m - 2$ registers, while process j writes its identifier into two registers. At that point, just before process j writes "waiting" into the two registers it owns, process i enters and exits its CS, releases the $m - 2$ registers it owned and then attempts to enter its CS again. Process i reads all the registers, finds out that no process is waiting, and is ready to try to own $m - 2$ free registers. Now, process j continues to write "waiting" into the two registers it owns, finds out that all the other $m - 2$ registers are free, and enters its CS. Process i is now scheduled, owns the $m - 2$ free registers and enters its CS, violating the mutual exclusion requirement.

It is easy to resolve this race condition while satisfying only deadlock-freedom. Resolving it while still satisfying starvation-freedom is more challenging. Our solution is as follows: the first thing that process i is doing (upon entering its entry code) is to own one register and only then check whether j is waiting. This will guarantee that either,

1. process i notices that j is waiting, in which case i releases its owned register and waits until no register has the value “waiting,” letting the waiting process enter its CS first; or
2. process i notices that no process is waiting, in which case, after j writes “waiting” into its owned registers; it will find out that not all the other registers are free, and will wait for process i to enter its CS first.

This solution resolves the race condition.

There is one additional thing that needs to be explained. Why do we reserve *two* registers for the waiting process, not just one? The answer is that by reserving two registers, when “waiting” is written into two registers by some process, at *least one* of the two will not be overwritten by the other process. Consider the following scenario. Let r_1, r_2 and r_3 be free registers. Process j writes into r_1,r_2 and is ready to write into r_3. Process i writes into all the $m-3$ other registers and is ready to write into r_3. Process j writes into r_3, finds out it is a loser (because $m \geq 7$), releases r_1, and writes “waiting” into r_2 and r_3. Now, process i is activated and writes into r_3, leaving only one register with the value “waiting.”

Finally, we explain why the algorithm does not work when $m = 5$ (or when $m = 3$). Assume $m = 5$. Let r_1, r_2, r_3, r_4 and r_5 be the five free registers. Consider the following scenario. Process i writes into r_1 and r_2 and is ready to write into r_3. Process j writes into r_5 and r_4 and is ready to write into r_3. Process i writes into r_3, finds out that it owns $m-2$ registers, and enters its CS. Then, Process j writes into r_3, finds out that it owns $m-2$ registers, and enters its CS, violating the mutual exclusion requirement. We point out that, in contrast, there is a symmetric memoryless deadlock-free mutual exclusion algorithm for two processes using $m \geq 3$ anonymous registers when m is odd.

3.2 Correctness Proof

▶ **Lemma 3.** *The algorithm satisfies mutual exclusion.*

Proof. We assume to the contrary that both processes enter their CS simultaneously and show that this leads to a contradiction. There are two ways for a process to enter its CS: (1) by observing that it owns $m-2$ registers, and (2) by writing “waiting” into the registers it owns and then waiting until all the other registers have the value 0. So, four possible combinations exist for having two processes in their CS simultaneously. We show that none of them may happen. Let’s call the two processes p and q.

1. *Both processes observe that they own $m-2$ registers and enter their CS.* Once p observes that it owns $m-2$ registers and enters its CS, at most one of these $m-2$ registers may later be overwritten by q. Thus, while p is in its CS, q may own at most 3 registers. Since $m \geq 7$, q cannot observe that it owns $m-2$ registers and hence will not enter its CS, while p is in its CS – a contradiction.
2. *Both processes write “waiting” into the registers they own and later enter their CS.* This may happen only if p and q observe that each one of them owns less than half of the registers. However, if p observes that it owns less than half of the registers, it must be the case that q owns more than half of the registers and will not write “waiting” into the registers it owns – a contradiction.

Algorithm 1 A symmetric memoryless starvation-free mutual exclusion algorithm for two processes.

```
CODE OF PROCESS $i$                                                        // $i \neq 0$

Constant:
    $m$: an odd integer $\geq 7$                          // $m \geq 7$ is the # of shared registers
Shared variables:
    $p.i[1..m]$: array of $m$ anonymous registers, of type integer + the symbol "waiting," init. all 0
                                  // $p.i[j]$ is the $j^{th}$ register according to process $i$'s numbering
Local variables:
    $myview[1..m]$: array of $m$ variables, initially all 0
    $mycounter, j, k$: integer, initially 0
    $mygo$: boolean, initially $false$

                                                              //give priority to a waiting process
repeat $mycounter \leftarrow mycounter + 1$ until $p.i[mycounter] = 0$    //looking for a zero entry
$p.i[mycounter] \leftarrow i$                                                  //own one register
for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for                   //read the shared array
if $\exists j \in \{1, ..., m\} : myview[j] = waiting$ then                 //is other process waiting?
    if $p.i[mycounter] = i$ then $p.i[mycounter] \leftarrow 0$ end if         //release owned register
    repeat                                                    //the other process is waiting
        for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for               //read the shared array
    until $\forall j \in \{1, ..., m\} : myview[j] \neq waiting$              //wait for CS to be released
end if

repeat                                                               //try to own $m - 2$ registers
    for $k = 1$ to $m$ do                                                   //access the $m$ registers
        if $p.i[j] = 0$ then                                                  //try to own one more
            for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for           //read the shared array
            if $i$ appears in less than $m - 2$ of the entries of $myview[1..m]$ then
            $p.i[j] \leftarrow i$ end if end if                                               //own one more
    end for
                                                                                   //lose or win?
    for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for                   //read the shared array
    if $i$ appears in less than $\lceil m/2 \rceil$ of the entries of $myview[1..m]$ then            //lose
        $mycounter \leftarrow 0$
        for $j = 1$ to $m$ do if $p.i[j] = i$ then                     //release all owned registers
            if $mycounter = 2$ then $p.i[j] \leftarrow 0$                               //except two of them
                else $p.i[j] \leftarrow waiting$; $mycounter \leftarrow mycounter + 1$ end if     //signal waiting
        end for
        repeat                                                        //wait for CS to be released
            for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ od                  //read the shared array
        until $\forall j \in \{1, ..., m\} : myview[j] \in \{0, waiting\}$        //no sign from other process
        $mygo \leftarrow true$                                                          //may enter CS
    end if
until $i$ appears in $m - 2$ of the entries of $myview[1..m]$ or $mygo = true$
critical section
                                                              //release all owned shared registers
if $mygo = true$ then for $j = 1$ to $m$ do if $p.i[j] = waiting$ then $p.i[j] \leftarrow 0$ end if end for
    else for $j = 1$ to $m$ do if $p.i[j] = i$ then $p.i[j] \leftarrow 0$ end if end for
end if
set all local variables to their initial values
```

3. *p observes that it owns $m-2$ registers and enters its CS, while q writes "waiting" into the registers it owns and* later *enters its CS while p is still in its CS.* Once p enters its CS, all the registers it owns are not free. So, q will not be able to proceed since not all the registers it does not own are free as required – a contradiction.
4. *p writes "waiting" into the registers it owns, waits until all the other registers have the value 0 and enters its CS. q observe that it owns $m-2$ registers and enters its CS, while p is still in its CS.* Since, after writing "waiting," p must observe that all the registers it does not own have the value 0 before it may enter its CS; it must be the case that q has written into one of the registers (line 2), after p has written "waiting." Thus, q will notice (line 4) that p is waiting and will not proceed to its CS – a contradiction. ◀

▶ **Lemma 4.** *The algorithm satisfies starvation-freedom.*

Proof. There are three loops where a process can get stuck.

- **Loop 1:** The repeat loop at lines 6-8, where a process waits until none of the registers is set to waiting.
- **Loop 2:** The inner repeat loop at lines 24-26, where a (waiting) process waits until all the registers are set to 0 or waiting.
- **Loop 3:** The outer repeat loop at lines 10-29 where a process waits until it either owns $m-2$ registers or its local register *mygo* is set to *true*.

We show that a process cannot get stuck (i.e., loop forever) in any of these loops, which implies starvation-freedom. Let's call the processes p and q.

Assume p is waiting in loop 1, and cannot proceed. This means that (1) p is not owning any of the registers, and (2) that q has set at least one of its owned registers to "waiting" (line 22). By the time q reaches loop 3 (line 24), all its owned registers are set "waiting," and all the other registers are free. Thus, q will exit the loop, set *mygo* to *true* and enter its CS. Later, in its exit code q will release its owned registers and return to its remainder region. From that point on (even if q will try to enter its CS again), as long as p is waiting in loop 1, no register will be set to waiting. This means that the condition in line 8 is evaluated to *true* and thus p can proceed beyond loop 1.

Assume p is waiting in loop 2, and cannot proceed. This means that (1) p has set the (one or two) registers it owns to "waiting" and is not trying to own more registers, and (2) q owns at least one register. There are two cases to consider:

- q notices (in line 4) that p is waiting, in which case q releases its owned register and waits until no register has the value "waiting," letting the waiting process p proceed beyond loop 2, setting *mygo* to *true* and proceed beyond loop 3.
- q notices that no process is waiting, in which case, after p writes "waiting" into its owned registers and waits in loop 2, there is nothing that prevents q from owning $m-2$ registers and entering its CS. Later, in its exit code q will release its owned registers and return to its remainder region. If q does not try to enter its CS again, then p can proceed beyond loop 2 and loop 3. If q does try to enter its CS again, then q might acquire (line 3) one register before p notices that the register is free; however, q will later notice (in line 4) that p is waiting, in which case q will release its owned register and waits until no register has the value "waiting," letting the p proceed beyond loop 2 and loop 3.

Assume p is looping in loop 3, never waits in loop 2, and cannot proceed. Since m is odd, this means that p owns more than half of the registers, while q holds less than half of the registers. Thus, q will eventually release the registers it owns, except at most two of them. This will enable p to acquire $m-2$ registers and proceed beyond loop 3. ◀

RMR complexity. An operation that a process performs on a memory location is considered a remote memory reference (RMR) if the process cannot perform the operation locally on its cache and must transact over the multiprocessor's interconnection network to complete the operation. RMRs are undesirable because they take long to execute and increase the interconnection traffic. Our algorithm achieves the ideal RMR complexity of $O(m)$ for cache coherent machines. (Distributed Shared Memory machines are irrelevant for anonymous shared memory systems.) This means that a process incurs $O(m)$ number of RMRs to satisfy a request (i.e, to enter and exit the critical section once). It follows from Observation 12 (Section 4) that this bound is tight.

4 An impossibility result

In the previous section, we have shown that, for two processes, it is possible to design a symmetric memoryless starvation-free mutual exclusion algorithm. Next, we show this is impossible for three or more processes.

▶ **Theorem 5.** *There is no symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using (any number of) anonymous registers.*

The main argument is that, under the appropriate assignment of names to registers, there is a way to run two processes when they are in their remainder section, requiring them to write to all registers before one of them can enter its critical section. This is essentially accomplished by renaming unwritten registers on the fly so that if the two processes are about to write to the same unwritten register for the first time, then they end up writing to two distinct, unwritten ones. Thus, it is possible to hide all the write operations of a third process, which will prevent it from ever entering its critical section.

4.1 Basic definitions and observations

We first prove some basic (but general) observations regarding the mutual exclusion problem. All the lemmas and definitions in Subsection 4.1 refer to one arbitrary deadlock-free mutual exclusion algorithm for n processes using read/write registers. Here, we do not need to assume that the algorithm is starvation-free, symmetric, or memoryless, nor do we need to assume that the registers are anonymous.

We will use the following notions and notations. An *event* corresponds to an atomic step performed by a process. An algorithm's (global) state is entirely described by the values of the (local and shared) registers and the values of the program counters of all the processes. A *run* is a sequence of alternating states and events (also called steps). For the purpose of the impossibility proof, it is more convenient to define a run as a sequence of events omitting all the states except the initial state. Since the events and the initial state uniquely determine the states in a run, no information is lost by omitting the states.

Each event in a run is associated with a specific process that is *involved* in the event. We will use x, y, and z to denote runs. When x is a prefix of y, we denote by $(y - x)$ the suffix of y obtained by removing x from y. Also, we denote by $x; seq$ the sequence obtained by extending x with the sequence of events seq. Processes are *deterministic*; that is, for every two runs $x; e$ and $x; e'$ if e and e' are events by the same process, then $e = e'$.

We will often use statements like "in run x process p is in its remainder", and implicitly assume that there is a function that for any run and process, lets us know whether a process is in its remainder, entry code, critical section, or exit code. Also, saying that an extension y of x involves only process p, means that all events in $(y - x)$ involve only process p. Finally, by a run we always mean a finite run, by a register we mean a shared register, and by the value of register r in run x, we always mean, the value of r at the end of x. Our first definition captures when two runs are indistinguishable to a given process.

▶ **Definition.** *Run x **looks like** run y to process p, if the subsequence of all events by p in x is the same as in y, and the values of all the registers in x are the same as in y.*

The looks like relation is an equivalence relation.[1] The next step by a given process always depends on the process's previous steps and the registers' current values. The previous steps uniquely determine whether the next step is a read or a write. The current values of the registers determine what value will be read in case of a read step. If two runs look alike to process p, then the next step by p in both runs is the same.

▶ **Lemma 6.** *Let x be a run which looks like run y to every process in a set P. If z is an extension of x which involves only processes in P, then $y;(z-x)$ is a run.*

Proof. By a simple induction on k – the number of events in $(z-x)$. The basis when $k=0$ holds trivially. We assume the lemma holds for $k \geq 0$ and prove for $k+1$. Assume that the number of events in $(z-x)$ is $k+1$. For some event e, it is the case that $z=z';e$. Since the number of events in $(z'-x)$ is k, by the induction hypothesis $y'=y;(z'-x)$ is a run. Let $p \in P$ be the process which is involved in e. Then, from the construction, the runs z' and y' look alike to p, which implies that the next step by p in both runs is the same. Thus, since $z=z';e$ is a run, also $y';e=y;(z-x)$ is a run. ◀

We next define the notion of a hidden process.[2] Intuitively, a process is hidden in a given run, if all the steps it has taken since the last time it has been in its remainder, communicate no information to the other processes. We say that a write event e_1 is *overwritten* by event e_2 in a given run r if e_2 is a write event that happens after e_1 in r, and both e_1 and e_2 are writing events to the same register.

▶ **Definition.** *For process p and run z, let z' be the longest prefix of z such that p is in its remainder in z'. Process p is **hidden** in run z if each event which p is involved in $(z-z')$ is either: a read event, or a write event that is overwritten (in z) before any other process has read the value written.*

We notice that a process is not hidden if it is involved in a write event that is not later overwritten, even if the write does not change the current value of a register. Also, if a process is in its remainder in z then it is hidden in z, and thus initially, all the processes are hidden. A hidden process looks just like a process halted in its remainder, and hence no process can wait until a hidden process takes a step.

▶ **Lemma 7.** *If a process p is in its critical section in run z then p is not hidden in z.*

Proof. Assume to the contrary that process p is hidden and is in its critical section in run z. Let z' be the longest prefix of z such that p is in its remainder in z'. Since p is *hidden* in run z, it is possible to remove from z all the events in which p is involved in $(z-z')$ and get a new run y. The run y looks like z to all processes other than p, and p is in its remainder in y. By the deadlock-freedom property, there is an extension of y that does not involve p in which some process $q \neq p$ enters its critical section. Since y looks like z to all processes

---

[1] The term *looks like*, adopted from [13], is also called *indistinguishable* in the literature.
[2] The notion of a hidden process was first defined in [3].

other than p, by Lemma 6, a similar extension exists starting from z. That is, q can enter its critical section in an extension of z, while p is still in its critical section. However, this violates the mutual exclusion property. ◀

It follows from Lemma 7 that a process must write before it enters its critical section.

4.2 Anonymity

The lemma in Subsection 4.2 refers to one arbitrary deadlock-free mutual exclusion algorithm for n processes using *anonymous* registers. Here, we do not need to assume that the algorithm is starvation-free, symmetric, or memoryless. We denote by e_p a (read or write) event which involves process p. When x is a run and π_p is a naming assignment of p, we denote by

- $x[p, r_i \leftrightarrow r_j]$ the sequence obtained by replacing every read event of register r_i by process p in x with a read event of register r_j by p which returns the same value (as the event of reading r_i), and vice versa.
- $\pi_p[r_i \leftrightarrow r_j]$ is the naming assignment where, (1) $\pi_p[r_i \leftrightarrow r_j](r_i) = \pi_p(r_j)$; (2) $\pi_p[r_i \leftrightarrow r_j](r_j) = \pi_p(r_i)$; and (3) for every $k \notin \{i, j\}$, $\pi_p[r_i \leftrightarrow r_j](r_k) = \pi_p(r_k)$.

Recall that all the anonymous registers are initialized to the same value.

▶ **Lemma 8.** *Let x be a run, p a process, π_p^x the naming assignment of p (used in the run x), and r_i and r_j two registers that were never written (by any process) in x. Then, $y = x[p, r_i \leftrightarrow r_j]$ is a run, where the naming assignment for p (used in y) is $\pi_p^x[r_i \leftrightarrow r_j]$, and for the other processes the naming assignments are the same as in x. Furthermore, if $x; e_p$ is a run where e_p is an event of writing the value v into r_i then $y; e'_p$ where e'_p is an event of writing the value v into r_j is also a run.*

Proof. Since r_i and r_j were never written in x, a read event by p or any other process of each of those registers in x would return the initial value. So, swapping the read events of r_i and r_j by process p, may only affect process p.

As for p, before its first event in x, the adversary has fixed, for process p, a naming assignment $\pi_p^x : \{r_1, ..., r_m\} \rightarrow \{r_1, ..., r_m\}$ of the registers such that, for process p, the j'th anonymous register is $\pi_p^x(r_j)$. In particular, when process p accesses its j'th anonymous register, it accesses $\pi_p^x(r_j)$. Let us define the permutation $\pi_p^y : \{r_1, ..., r_m\} \rightarrow \{r_1, ..., r_m\}$ as follows, (1) $\pi_p^y(r_i) = \pi_p^x(r_j)$; (2) $\pi_p^y(r_j) = \pi_p^x(r_i)$; and (3) for every k where $k \notin \{i, j\}$, $\pi_p^y(r_k) = \pi_p^x(r_k)$. That is, whenever p accessed r_i before it will now access r_j and vice versa. Now, consider a run in which the processes are scheduled exactly in the order as they are scheduled in x, the naming assignment fixed (by the adversary) for p is π_p^y, and the naming assignments (in y) of the other processes are as in x. By construction, the resulting run is run y. Furthermore, if the next event by p in x is of writing the value v into r_i then, since $\pi_p^y(r_i) = \pi_p^x(r_j)$, the next event by p in y is of writing the value v into r_j. ◀

4.3 The notions of a symmetric run and a symmetric state

A (global) state is entirely described by the values of the local and shared registers and the values of the program counters of all the processes.

Intuitively, a state σ is symmetric w.r.t. two processes if the "subjective views" of the processes at σ are the same. In the standard non-anonymous model, once two (symmetric) processes write their ids (one after the other) into the same register, say r_1, their views after these two writes are completed are no longer the same, if they inspect the current state, one process will see that the value of r_1 equals its id, while the other will see that the value is different from its id. In the anonymous model, it is possible for each process to see that the value of the first register according to its naming assignment equals its id.

Below we define this notion more formally. Let m be the number of registers; let $val_\sigma(r)$ be the value of register r in state σ, and assume that the names of the local variables of the processes are the same. To distinguish the local variables of the different (symmetric) processes, we will add the process id as a subscript to the variable names.

▶ **Definition.** *Let σ be a state and let π_p and π_q be the naming assignments of p and q, respectively. State σ is* ***symmetric*** *w.r.t. p and q and their naming assignments π_p and π_q, if for every $1 \leq k \leq m$ either,*

- $val_\sigma(\pi_p(r_k)) = val_\sigma(\pi_q(r_k))$ *and* $val_\sigma(\pi_p(r_k)) \notin \{p, q\}$, *or*
- $val_\sigma(\pi_p(r_k)) = p$ *and* $val_\sigma(\pi_q(r_k)) = q$, *or*
- $val_\sigma(\pi_p(r_k)) = q$ *and* $val_\sigma(\pi_q(r_k)) = p$.

Furthermore, in σ, for every local variable, say local, either (1) $local_p = local_q$ and the value of $local_p$ is not in $\{p, q\}$, or (2) $local_p = p$ and $local_q = q$, or (3) $local_p = q$ and $local_q = p$.

▶ **Definition.** *Run x is* ***symmetric*** *w.r.t. p and q and their naming assignments π_p and π_q, if the state at the end of x is symmetric w.r.t. p and q and their naming assignments π_p and π_q.*

The following lemma follows immediately from the definitions of a symmetric run and a symmetric state.

▶ **Lemma 9.** *When processes p and q are symmetric (and deterministic), if run x is symmetric w.r.t. p and q and their naming assignments π_p and π_q, then*

1. *either the next step of both processes is a read or the next step of both is a write;*
2. *if p accesses $\pi_p(r_k)$ in its next step, then q accesses $\pi_q(r_k)$ in its next steps;*
3. *if x is extended by a read event of p followed by a read event of q, then the resulting run is symmetric w.r.t. p and q and their naming assignments;*
4. *if x is extended by a write event of p followed by a write event of q then the resulting run is symmetric w.r.t. p and q and their naming assignments, provided that p and q do not write into the same physical location; and*
5. *p and q cannot be in their critical sections at (the end of) x.*

▶ **Lemma 10.** *In a symmetric deadlock-free mutual exclusion algorithm for $n \geq 2$ processes using anonymous registers, the initial state where all the processes are in their remainder sections is symmetric w.r.t. every two processes and their naming assignments.*

Proof. The proof follows immediately from the fact that in anonymous shared memory, all the anonymous registers are initialized to the same value; and that when the processes are symmetric, the local variables of the different processes have the same names and (local variables with the same names) are initialized to the same values. ◀

To simplify the presentation, for the rest of the section, by a *symmetric run* (resp. *symmetric state*), we always mean a symmetric run (resp. symmetric state) w.r.t. to every two processes and their naming assignments.

4.4 Symmetry and anonymity

The following lemma in Subsection 4.4 refers to one arbitrary symmetric deadlock-free mutual exclusion algorithm for $n \geq 2$ processes using m *anonymous* registers. Here, there is no need to assume that the algorithm is starvation-free or memoryless.

A *quiescent* state is one in which all the processes are in their remainder sections. A memoryless algorithm is an algorithm that has exactly one quiescent state (which is the initial state). When the possible number of quiescent states of a symmetric starvation-free mutual exclusion algorithm is more than one, by Lemma 10, the initial (quiescent) state is symmetric w.r.t. every two processes and their naming assignments. However, for a non-memoryless algorithm, except for the initial state, the other quiescent states are not necessarily symmetric.

▶ **Lemma 11** (main technical lemma). *For every two processes p and q, and every symmetric quiescent state σ, there exist naming assignments π_p and π_q for p and q, and a run ρ with the following properties,*

1. *ρ starts from the state σ and ends in some quiescent state,*
2. *during ρ, only p and q take steps, and they enter and exit their critical sections once,*
3. *during ρ, each one of the two processes writes into $\lceil m/2 \rceil$ different registers, and*
4. *during ρ, each one of the m anonymous registers is written at least once.*

Proof. It follows from Theorem 1 that (when using anonymous registers) any symmetric deadlock-free mutual exclusion algorithm for two processes must use an odd number of anonymous registers. So, for the rest of this proof, we assume that m is odd.

We prove the lemma by running the two processes p and q, starting from the (symmetric) state σ, keeping the run symmetric (i.e., without breaking symmetry). As long as symmetry is not broken, by Lemma 9, none of the processes can enter its CS, because if it does, then the other process may enter its CS as well, violating the mutual exclusion requirement. Each time two more registers are written until only one is left. At that point, the two processes try to write this last register. Only when this last register is written is symmetry broken, all the m anonymous registers are written, and we are done.

Initially, the adversary fixes for process p, the identity permutation $\pi_p(r_k) = r_k$ (where $1 \leq k \leq m$), and fix for q its reverse permutation $\pi_q(r_k) = r_{m-k+1}$. For the sake of explanation, we arrange the registers in pairs, where the k^{th} pair is $(\pi_p(r_k), \pi_q(r_k))$, where $1 \leq k \leq m$. In this pairing, each pair includes two different registers except the $\lceil m/2 \rceil$ pair in which the register $r_{\lceil m/2 \rceil}$ appears twice. Each other register appears in two pairs, and the other register is the same in those two pairs. For example, when $m = 5$, the pairs are (r_1, r_5), (r_2, r_4), (r_3, r_3),(r_4, r_2), and(r_5, r_1).

Next, we run the processes in *lock-steps*. Each lock-step includes a step by p followed by a step of q. We observe that, as long as the constructed run is symmetric, in one lock-step, if p accesses r_i and q accesses r_j, then (1) the pair (r_i, r_j) appears in the pairing described above, and (2) by Lemma 9, both processes either read r_i and r_j or write into them. Because of the initial symmetry, none of the processes may enter its CS without writing first. (this also follows from Lemma 7).

The run ρ is constructed by iteratively executing the following procedure until all the m registers are written at least once, as required. Let π_p and π_q denote the current naming assignments of p and q. Let us denote by x_{i-1}, the run constructed so far, before the beginning of the i^{th} iteration. As we will see, by construction, x_i will be a symmetric run for all $i \geq 0$. By assumption, the quiescent state σ is symmetric, and thus, the run x_0, where all the processes are still in their remainder sections, is symmetric.[3]

Iteration $i \geq 1$ begins here. As explained, x_{i-1} is the symmetric run constructed so far. First, p and q run (in lock-steps) until they are ready to write into r_i and r_j, respectively. Recall that by Lemma 9, either both steps are read or both are write, in each lock-step.

---

[3] We notice that in the special case where the algorithm is memoryless, there is no need to assume that σ is symmetric, as by Lemma 10, the single initial (quiescent) state σ is symmetric.

Thus, to break symmetry they must eventually try to write. Let's denote by y_{i-1} the run just before the two writes. Clearly, if x_{i-1} is symmetric, then so is y_{i-1}. There are four possible cases,

1. r_i and r_j are different registers. In this simple case, we let both processes complete their write operations. The run x_i is the run just after these two writes. If x_{i-1} is symmetric, then so is x_i. The i^{th} iteration completes here, and we are ready to start the next $i+1$ iteration.
2. r_i and r_j refer to the same register, which has been written before. The construction ensures this situation never happens (see Case 4 below).
3. r_i and r_j refer to the same register which has not been written before, and all other $m-1$ registers have already been written. In this case, we let both processes complete their write operations. At this point, all the m registers have been written as requested. So we let the processes continue running until, as guaranteed by the deadlock-freedom property, each one of them will eventually enter its CS (but not simultaneously), return to its remainder section, and we are done. The constructed run is ρ, and **the construction of the run ρ terminates here.**
4. r_i and r_j refer to the same register which has not been written before, and *not* all other $m-1$ registers have already been written. This is the more challenging case. For some k, let $\pi_p(r_k)$ and $\pi_q(r_k)$ be two registers (different from r_i) that have not been written so far (we do not care whether $\pi_p(r_k)$ and $\pi_q(r_k)$ have or have not been read so far). By Lemma 8, $z^1_{i-1} = y_{i-1}[p, r_i \leftrightarrow \pi_p(r_k)]$ is a legal run, assuming π_p is replaced with $\pi_p[r_i \leftrightarrow \pi_p(r_k)]$. Furthermore, by applying lemma 8 again, $z^2_{i-1} = z^1_{i-1}[q, r_j \leftrightarrow \pi_q(r_k)]$ is a legal run, assuming π_q is replaced with $\pi_q[r_i \leftrightarrow \pi_q(r_k)]$. Since r_i, $\pi_p(r_k)$ and $\pi_q(r_k)$ have never been written yet, if x_{i-1} is symmetric, so is z^2_{i-1}.
   Furthermore, at (the end of) z_2, p and q are ready to write into the two registers $\pi_p(r_k)$ and $\pi_q(r_k)$, respectively. We now continue with run z^2_{i-1}, and let both processes complete their write operations into $\pi_p(r_k)$ and $\pi_q(r_k)$. Again symmetry is preserved. The resulting run, after the writes is x_i with which we continue to the next iteration. It is important to notice that in this case (unlike in the other two cases) x_i is *not* an extension of x_{i-1}, it is a completely new run. In addition, in the next round, which starts with x_i, we use the (updated) naming assignments $\pi_p[r_i \leftrightarrow \pi_p(r_k)]$ and $\pi_q[r_j \leftrightarrow \pi_q(r_k)]$ for processes p and q. As explained x_i is symmetric w.r.t. p and q and their updated naming assignments.

In case 4, we switch to a new run and new naming assignments. We emphasize that, during a specific run, we do not change the naming assignments associated with that run. We change them only when we switch to a completely new run. The run ρ that we end up with at the end of the construction uses fixed naming assignments for p and q from the beginning to the end. This whole construction can be viewed as searching for a run ρ and fixed naming assignments π_p and π_q for which *Case 4* will never happen, and this is exactly what we end up with. That is, once run ρ is constructed with its associated naming assignments for p and q, if we consider execution of run ρ starting from σ with the associated naming assignments, then, by construction, the only time p and q will try to write into the same register, would be *after* all other $m-1$ registers have already been written. ◀

▶ **Observation 12.** *The RMR complexity of every symmetric deadlock-free mutual exclusion algorithm for $n \geq 2$ processes using m anonymous registers is at least $\lceil m/2 \rceil$.*

Proof. Let σ be the initial state. By Lemma 10, the initial state σ is symmetric. By Lemma 11, there exists a run which starts from σ, in which exactly two processes participate during which they enter and exit their critical sections once, and each one of the two processes writes into $\lceil m/2 \rceil$ different registers. The result follows. ◀

4.5 Proof of Theorem 5

Proof. Assume to the contrary that there is a symmetric memoryless *starvation-free* mutual exclusion algorithm for $n \geq 3$ processes using anonymous registers. Let's call this algorithm A. Let us denote by σ the single possible initial state. By Lemma 10, σ is symmetric. Let p_1, p_2 and q be three processes. Using Lemma 11, we will reach a contradiction by hiding all the write operations of q, which, by Lemma 7, will prevent q from ever entering its critical section.

This is done as follows. Assume all the processes are in their remainder sections; thus, the current state is σ. Now, process q tries to enter its critical section. Before doing so, q should execute its entry section which must, by Lemma 7, involve at least one write operation. So, we run q alone until it is about to execute its first write operation. Let's call the first register that q is about to write register r_1. Since q has not modified any register yet, all the processes except q cannot distinguish the current state from the state σ where all the processes (including q) are still in their remainder sections. Thus, by Lemma 6 and Lemma 11, there is an extension of the current run which involves only p_1 and p_2, in which p_1 and p_2 enter and then exit their critical sections once, and in that extension, each one of the anonymous registers is written (by either p_1 or p_2) at least once. We slightly modify this extension of p_1 and p_2 by stopping them just before writing into register r_1; let q complete a write operation into r_1; then let p_1 and p_2 overwrite the value written by q, and continue until p_1 and p_2 return to their remainder sections.

Since the write of q into r_1 was immediately overwritten, q is hidden. Hence all the processes except q cannot again distinguish the current state from the state σ where all the processes (including q) are still in their remainder sections. Notice that here we use the assumption that the algorithm is memoryless, and this will enable us to use Lemma 11 repeatedly. It is important to understand that since we have already used Lemma 11 once in the proof, the naming assignments for processes p_1 and p_2 have been fixed, and these assignments can not be changed after that. However, because of the memoryless assumption, this does not prevent us from applying Lemma 11 again to σ using processes p_1 and p_2.

We notice that *without* the memoryless assumption, after p_1 and p_2 return to their remainder sections, it is no longer necessarily true that all the processes except q cannot again distinguish the current state from the state σ. This is so because there might be another state σ' where all the processes are in their remainder sections. Hence, in such a case (without the memoryless assumption), it would not be possible to apply Lemma 11 again.

By Lemma 7, at this point, since q is hidden, q must write again before it may enter its critical section. So, we run q alone until it is about to execute its second write operation. Let's call the register that q is about to write register r_2. (Register $r2$ might denote a different or the same register as r_1.) As before, all the processes except q cannot distinguish the current state from the state σ. Thus, again by Lemma 11, there is an extension of the current run that involves only process p_1 and p_2, in which p_1 and p_2 enter and then exit their critical sections once, and in that run each one of the anonymous registers is written at least once. We slightly modify this extension of p_1 and p_2 by stopping them just before writing register r_2; let q complete a write operation into r_2; then let p_1 and p_2 overwrite the value written by q, and continue until they return to their remainder sections. Since the write of q into r_2 was also immediately overwritten, q is still hidden, and thus all the processes except q cannot distinguish the current state from σ.

We can apply the above procedure as often as necessary, hiding all the write operations of q, which by Lemma 7, prevents q from ever entering its critical section. Thus, algorithm A does not satisfy starvation-freedom as promised. A contradiction. ◀

4.6 A generalization

For a non-memoryless algorithm, except for the initial state, the other quiescent states are not necessarily symmetric. Under the assumption that all the quiescent states are symmetric, it is possible to prove the following generalization of Theorem 5.

▶ **Theorem 13.** *There is no symmetric starvation-free mutual exclusion algorithm, with at most s quiescent states, for $n \geq 2s+1$ processes using (any number of) anonymous registers, assuming all the quiescent states are symmetric.*

Proof. We slightly modify the proof of the impossibility result in Subsection 4.5. We assume to the contrary that there is a symmetric *starvation-free* mutual exclusion algorithm for $n \geq 2s+1$ processes using anonymous registers. With each quiescent state σ_i we associate two processes p_1^i and p_2^i. Let q be a process not associated with any quiescent state. Using the above-modified version of Lemma 11, we can reach a contradiction by hiding all the write operations of q, which, by Lemma 7, will prevent q from ever entering its critical section.

This is done as follows. Assume that all the processes are in their remainder sections, and the current state is σ_i which is assumed to be symmetric (this is required for applying the modified version of Lemma 11). Now, process q tries to enter its critical section. Before doing so, q must execute at least one write operation. So, we run q alone until it is about to execute a write operation. Let's call the register that q is about to write, register r. Since q has not modified r yet, all the processes except q cannot distinguish the current state from the quiescent state σ_i. Thus, there is an extension of the current run, which involves only p_1^i and p_2^i, in which p_1^i and p_2^i enter and then exit their critical sections once, and in that extension, each one of the anonymous registers is written at least once. We slightly modify this extension by stopping p_1^i and p_2^i just before writing into register r; let q complete a write operation into r; then let p_1^i and p_2^i overwrite the value written by q, and continue until p_1^i and p_2^i return to their remainder sections.

Since the write of q into r was immediately overwritten, q is still hidden, and thus all the processes except q cannot distinguish the current state from one of the s quiescent states. We can apply the above procedure as often as necessary, hiding all the write operations of q, which by Lemma 7, prevents q from ever entering its critical section. Thus, the algorithm does not satisfy starvation-freedom as promised – a contradiction. ◀

5 Discussion

We have shown that, while for two processes, it is possible to design a symmetric memoryless starvation-free mutual exclusion algorithm using anonymous registers, this is impossible for three or more processes. These results imply that, while there is no separation between deadlock-freedom and starvation-freedom for two processes, such a separation between deadlock-freedom and starvation-freedom exists for three or more processes. Thus, in anonymous shared memory systems where process symmetry is assumed, it is impossible always to ensure individual progress in situations where global progress is possible. This is the first known case of fault-free systems, demonstrating a separation between individual and global progress.

References

1 Z. Aghazadeh, D. Imbs, M. Raynal, G. Taubenfeld, and Ph. Woelfel. Optimal memory-anonymous symmetric deadlock-free mutual exclusion. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 157–166, 2019.

2 H. Attiya, A. Castañeda, D. Hendler, and M. Perrin. Separating lock-freedom from wait-freedom at every level of the consensus hierarchy. *Journal of Parallel and Distributed Computing*, 163:181–197, 2022. Conf. version appeared in PODC 2018.

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

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

5 M. Herlihy. Impossibility results for asynchronous PRAM. In *Proc. of the 3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 327–336, 1991.

6 M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

7 M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

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

9 S. Rashid, G. Taubenfeld, and Z. Bar-Joseph. The epigenetic consensus problem. In *28nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2021)*, pages 146–163, June 2021. *LNCS 12810.*

10 M. Raynal and G. Taubenfeld. Symmetry and anonymity in shared memory concurrent systems. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, 136, 2022. 17 pages.

11 M. Raynal and G. Taubenfeld. A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47–65, June 2022.

12 E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.

13 G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

14 G. Taubenfeld. Coordination without prior agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 325–334, 2017.

15 G. Taubenfeld. Anonymous shared memory. *J. ACM*, 69(4):1?30, August 2022.