A visit to mutual exclusion in seven dates

Michel Raynal, Gadi Taubenfeld

Version of Record: https://www.sciencedirect.com/science/article/pii/S0304397522001815
Manuscript_cb8f1d7bcade51be96e057c8dc7e1c41

A Visit to Mutual Exclusion in Seven Dates

Michel Raynal[†,⋆] Gadi Taubenfeld[‡]

[†]IRISA, Université de Rennes, 35042 Rennes, France
[⋆]Department of Computing, Polytechnic University, Hong Kong
[‡]Reichman University, Herzliya 46150, Israel

raynal@irisa.fr tgadi@idc.ac.il

Abstract

Mutual exclusion (mutex) is one of the most fundamental synchronization problems encountered in shared memory systems. It appears in all computer science first-degree curricula. This article presents nine mutex algorithms, each with its noteworthy features, spread over seven dates covering 1965-2020. Most of these algorithms are very well known and paved the way for new research directions. This article aims to present fundamental issues and basic principles that underlie the design of shared memory mutex algorithms in different contexts. So, differently from exhaustive surveys on shared memory mutex algorithms, it strives to give the reader a flavor of the many design facets of this still challenging problem.

Keywords: Anonymity, Asynchronous system, Atomic register, Concurrency, Deadlock-freedom, Mutual exclusion, Read/Write register, Read-Modify-Write register, Safety, Liveness, Safe register, Starvation-freedom.

1 Introduction

The *mutual exclusion* problem (mutex) guarantees mutually exclusive access to a single shared resource when several competing processes exist. The problem arises in operating systems, database systems, parallel supercomputers, and computer networks, where it is necessary to resolve conflicts resulting when several processes are trying to use shared resources, such as data items, files, discs, printers, etc. For example, the integrity of the data may be destroyed if two processes update a common file at the same time, and as a result, deposits and withdrawals could be lost, confirmed reservations might disappear, etc. In such cases, it is essential to allow at most one process to use a given resource at any given time. Mutual exclusion is of great significance since it lies at the heart of many synchronization problems in concurrent programming involving cooperation or competition issues.

The notion of sequential processes and the need to synchronize them –for cooperation or conflict resolution– was introduced by Edsger W. Dijkstra in the very early sixties (1962) [31], who introduced the mutual exclusion problem and its property-based formulation [32]. According to him, the first mutual exclusion algorithm (for two processes) was proposed by Dekker[1]. Since then, numerous solutions for the mutex problem have been proposed. Moreover, due to its importance and new hardware and software developments, new solutions to the problem are being designed all the time. The first book entirely devoted to mutual exclusion appeared in 1986 [72]. Numerous surveys devoted to mutual exclusion have been published. Among them, [9] (2003) covers the period 1996-2003. More recently (2015) [19] presents an in-depth study of the correctness and performance of an extensive set of mutual exclusion algorithms.

The present article uses an informal style to provide the reader with a short algorithmic-based visit of the mutual exclusion problem since its introduction in the early sixties. This visit is articulated around seven dates, each associated with a given computing model and one or two algorithms. Its aim is neither to focus on proof techniques of algorithms solving the mutex problem (which can be found in textbooks, surveys, or specific articles, e.g. [20, 24, 47, 50, 73, 82]) nor to be an in-depth presentation coming with experimental evaluations of numerous mutex algorithms (as nicely done in [19]). By taking a date-based historical view, the article aims to trace an evolution in the motivation for new mutex algorithms, which led to new principles in their design. Table 1 presents a synoptic picture of the visit.

Year	Main feature	Section	Reference
1965	Dijkstra's algorithm	3	[32]
1974	No underlying atomicity	4	[53]
1981-82	Simplicity and Efficiency	5	[50, 68]
1985	Real-time	6	[36]
1987	Adaptivity	7	[58]
1990-91	Local spinning	8	[10], [64]
2020	Full anonymity	9	[77]

Table 1: Global picture

Remark Due to subtle interferences, concurrent algorithms are prone to errors. Their exhaustive testing can be impossible and possibilities for model checking are limited. So, their correctness must be *proved*. The present article is on the design principles of mutex algorithms and not on associated proof techniques. However, some of the presented algorithms are proved and references to proofs of other algorithms are given.

[1]The reader interested in Dekker's algorithm will find an in-depth (both theoretical and practical) study of it in [20].

2 Preliminaries

2.1 Process and communication model

Except one, all the presented algorithms consider a system composed of n asynchronous sequential processes, denoted p_1, ..., p_n, which communicate by accessing shared registers. When needed, the integer i is used as the identity of p_i. *Asynchronous* means that each process progresses at its pace, which can vary with time and is unknown to the other processes. The processes are assumed to be failure-free. We consider four operations that the processes can use to access the shared registers as defined below. These operations are read, write, fetch&increment, swap and compare&swap.

Unlike the other algorithms, the algorithm presented in Section 6 considers a partially synchronous system. Based on the real-time passage, that system model assumes that two consecutive accesses to the shared memory by the same process are separated by at most Δ time units, Δ being an upper bound known by the processes.

Read/write registers A register accessed only by read and write operations is called a read/write register. Such a register is either a multi-writer multi-reader register (all the processes can read and write it, in short MWMR), or a single-writer multi-reader register (a single process can write and all the processes can read it, in short SWMR). Given an array $A[1..n]$ of SWMR registers, only p_i can write $A[i]$.

A read/write register is *atomic*, if while the read and write operations that access it can be concurrent, they appear as if they have been executed one after the other, this total order complying with their real-time order, and the projection of this sequence on each register satisfies the semantic of a sequentially accessed register (the value returned by the reading of a register R is the value written by the closest preceding write invocation (or the initial value if there is no such write).

A read/write register is *non-atomic* if it is not atomic. Informally, the weakest possible SWMR non-atomic register (also called *safe* register [56]) is where it is assumed only that a read not concurrent with any writes obtains the correct value. That is, the read should return the value written by the most recent write, or the initial value if no write had yet occurred. In the case where a read from a register is concurrent with some write into that register, the read may return an arbitrary value that matches the type of the register. The model in which the processes communicate only through atomic (resp. non-atomic) read/write operations is denoted atomic-RW (resp., non-atomic-RW).

Beyond read/write registers In addition to atomic read/write operations on registers, some processors provide users with "stronger" atomic operations such as the following ones, denoted $\mathsf{fetch\&increment}()$, $\mathsf{swap}()$, and $\mathsf{compare\&swap}()$ as defined below. The corresponding communication model is denoted atomic-RW+.

- Let v be the current value of register R. The invocation of $\mathsf{fetch\&increment}(R)$ by a process p_i writes $v + 1$ in R and returns the value v.
- Let v be the current value of register R. The invocation of $\mathsf{swap}(R, w)$ by a process p_i writes w in R and returns its previous value v.
- In the invocation of $\mathsf{compare\&swap}(R, old, new)$, R is a shared register, while old and new are two values. The value new is assigned to R only if $R = old$. In this case, the invocation returns `true`. Otherwise, the value of R remains unchanged, and the invocation returns `false`.

Notations Uppercase letters are used for the identifiers of shared registers. Lowercase letters and process index are used for the local variables of a process (e.g., aux_i is a local variable of p_i).

2.2 The Mutex Problem

Each process p_i has a special piece of code called a *critical section*, denoted cs_i. When process p_i is executing cs_i, no other process p_j can be simultaneously executing cs_j.[2] The mutual exclusion problem is to build two operations, denoted acquire() and release(), such that

- for each process p_i, instead of executing directly only cs_i, p_i must execute the pattern acquire(); cs_i; release(), and
- the following requirements must be satisfied:
  - Mutual exclusion (safety): no two processes are simultaneously inside their critical sections;
  - Deadlock-freedom (liveness): if one or more processes concurrently invoke acquire(), at least one of these invocations must eventually terminate.

The above liveness property does not state that, in the presence of concurrency, all invocations of acquire() terminate. This constitutes a stronger liveness property, namely

- Starvation-freedom: if a process invokes acquire(), its invocation must eventually terminate.

A Dijkstra's citation It is worth citing one of the very first sentences of E. W. Dijkstra's article introducing mutex [32]. This sentence looks like a prophecy, namely:

> "Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved."

The initial seed needed to implement mutual exclusion Mutex is a symmetry-breaking problem, in the sense that, all processes being "equal", at any time, one and only one process is allowed to be in the critical section. This means that an initial asymmetry seed must be present in the system to be able to solve mutex [21, 23, 80]. In most mutex algorithms, which are based on atomic read/write registers, this initial seed lies in the identities of the processes, which must be different and can be compared. Moreover, in the case of algorithms based on SWMR atomic registers, the identities are used to allow each process to write into its register.

Symmetric algorithms The notion of a *symmetric* algorithm was introduced in [80]. In a symmetric algorithm, the only way to distinguish two processes is by comparing their identities. This defines a specific programming type whose values can only be read, written, and compared. They cannot be used for anything else (e.g., they cannot address array entries). So, a symmetric algorithm cannot be based on SWMR atomic registers.

Moreover, a *symmetric with equality* algorithm is an algorithm in which the process identities can only be compared with equality – there is no notion of "smaller" or "greater" among them, it is only possible to check if two variables containing process identities are equal or not (or equal to a default value $\perp$ considered as a "nil" process identity).

Finally, an algorithm satisfies the *full symmetry* property when the processes are anonymous (they have no name, the same code with the same initialization). The interested reader will find in [79] more developments on the notions of symmetry and anonymity.

[2] Let us observe that this definition of mutex is very general. Taking $cs_i = cs_j = ... = cs$, this definition boils down to the mutex definition introduced in [32].

2.3 Lower bounds

As for any other problem, there are computability results and lower bounds associated with mutex. These bounds are on the number of steps and the number of shared registers needed to solve n-process mutex on top of read/write registers.

- *Space*: Assuming n processes with different and comparable identities, n atomic MWMR bits are necessary and sufficient to solve deadlock-free mutex (but not starvation-freedom) [23].

  Differently, if the atomic register can also be atomically accessed by a more powerful operation such as compare&swap, a single register is sufficient for deadlock-free mutex. More results on mutex-related memory bounds, when using powerful operations, can be found in [22].

- *Time*: There is no two (or more) process mutual exclusion algorithm, with an upper bound on the number of times a winning process may need to access the shared memory to enter its critical section in the presence of contention [6] (see also [82], page 119).

  That is, in the presence of contention the adversary can schedule the contending processes in such a way that the winning process will have to *spin* (busy-wait) on a register, until some other process terminates the spin with a single write operation. Unfortunately, under contention, such spinning may generate lots of traffic on the interconnection network between the process and the memory. Hence, by consuming communication bandwidth spin-waiting by some process can slow other processes.

3 1965: Dijkstra's n-Process Mutex Algorithm

This algorithm was introduced in [32]. In this one-page article, E. W. Dijkstra defined the mutex problem in terms of properties (that any algorithm solving it must satisfy), described an associated algorithm, and gave proof of it!

Shared atomic read/write registers The communication model is atomic-RW. It is made up of the following set of registers. $FLAG[1..n]$ is an array of SWMR Boolean registers, all initialized to `false`.

$NOTN[1..n]$ is an array SWMR Boolean registers, initialized to $[\texttt{true}, \cdots, \texttt{true}]$ ($NOTN$ means "not the next process to enter the critical section"). $NEXT$ is an MWMR register that contains the identity of the process that is considered to be the next to enter the critical section. Its initial value is arbitrary. Let us note that the invocation of $FLAG[NEXT]$ requires two consecutive (asynchronous) accesses to the shared memory.

The atomic Boolean $FLAG[i]$ is used by p_i to inform the other processes that it is currently competing for the critical section or executing code inside it. The atomic register $NEXT$ contains the identity of the next process to enter the critical section. The flag $NOTN[i]$ is raised by p_i when it is looping to become the next process to enter the critical section. A process terminates an operation acquire() or release() when it executes the associated return() statement.

Dijkstra's algorithm The algorithm is described in Fig. 1. When process p_i wants to enter the critical section, it raises its flag $FLAG[i]$ (line 1). It will lower it when it exits the critical section (line 10). After it raised $FLAG[i]$, process p_i enters a repeat loop whose aim is to guarantee that exactly one competing process will enter the critical section. There are two cases.

- If $NEXT \neq i$, p_i indicates to the other processes, it is not the next process (line 4) and strives to be it. To this end, if p_{NEXT} is not interested in the critical section (predicate $FLAG[NEXT] = \texttt{false}$), p_i competes to be the next (assignment $NEXT \leftarrow i$ at line 5)), and re-enters the repeat loop.

```
operation acquire() is
(1)   FLAG[i] ← true;
(2)   repeat
(3)     if (NEXT ≠ i)
(4)       then NOTN[i] ← true;
(5)            if (FLAG[NEXT] = false) then NEXT ← i end if
(6)       else NOTN[i] ← false;
(7)            if (∀ j ≠ i : NOTN[j]) then return() end if
(8)     end if
(9)   end repeat.

operation release() is
(10)  FLAG[i] ← false; NOTN[i] ← true; return().
```

Figure 1: Dijkstra's 1965 n-process mutex algorithm (code of process p_i) [32]

- If $NEXT = i$, p_i writes `false` in $NOTN[i]$ and terminates its invocation of acquire() if it sees that each other process is not claiming it the next to enter (line 7). Otherwise, p_i re-enters the repeat loop.

Proof of the safety property Let R be the region of code including line 7, the critical section code, and line 10. Let L_i be the set of process indexes k for which p_i has not yet verified $NOTN[k]$. Let us observe that, when two different processes p_i and p_j are in the region R, we have $i \in L_j \ \vee \ j \in L_i$, i.e, we have the global invariant (where $p_k \in R$ means that p_k is inside the region R):

$$\forall\, i, j : \ i \neq j : \ (\{p_i \in R) \wedge \ (p_j \in R) \Rightarrow (i \in L_j) \ \vee \ (j \in L_i).$$

We then have the following.

- If process p_i enters R while p_j in already in R, we have $j \in L_i$. If p_i reads $NOTN[j] = \texttt{false}$, it exits the region R (jumping to line 2). The same holds for p_j, which proves the invariant.
- If process p_i is inside the critical section, we have $L_i = \emptyset$, and consequently, due to the invariant, we have $i \in L_j$.

It follows that no two different processes can be concurrently inside their critical section.

Proof of the deadlock-freedom property Let us assume by contradiction that one or more processes invoke acquire() and none of them returns from its invocation. Due to lines 1 and 5, there is a finite time τ after which there is a competing process p_i such that the predicates $NEXT = i$ and $FLAG[i] = \texttt{true}$ remain forever true. After τ, p_i repeatedly executes only the lines 6-7, while each other competing process p_k executes only the lines 4-5, at which it assigns `true` to $NOTN[k]$. It follows that there is a finite time after which the predicate $(\forall\, j \neq i : \ NOTN[j])$ forever remains true. When this occurs, due to line 7, p_i terminates it invocation, which contradicts the initial assumption.

From deadlock-freedom to starvation-freedom While Dijkstra's mutex algorithm ensures deadlock-freedom, it does not ensure that all invocations of acquire() terminate. The interested reader will find in [51] (1966) a mutex algorithm, due to Knuth, that "extends" the previous mutex algorithm to obtain the starvation-freedom liveness property.

4 1974: When the Read/Write Registers Are Not Atomic

As with many other mutex algorithms, Dijkstra's algorithm allows us to define "big" atomic operations (whose codes define critical sections) from atomic read/write registers offered by the underlying hardware. This gives rise to the following question "Is it possible to implement mutex from registers that are not atomic?" This question was posed and solved by L. Lamport in 1974, who was looking for a *real solution* to the mutex problem, namely an algorithm building atomicity without relying on atomicity at a lower level [53].

The notion of a safe register Introduced by L. Lamport in [53], and then deeply investigated in [56, 57], a *safe* read/write register is an SWMR register satisfying the following very weak properties:

- When the write of a value v terminates, the register contains v.
- A read of the register that occurs while there is no concurrent write returns the current value of the register.
- A read of the register that occurs while there is a concurrent write returns *any* value that the register can contain.

To see the weakness of a safe register, let us consider a safe bit that contains 0. Suppose that a read is concurrent with a write of the value 1. The value returned by the read can be 0 or 1. Suppose now that a read is concurrent with a write of the value 0. As before, the value returned by the read can be 0 or 1. Hence, even if the value of the register has not changed, due to concurrency between reading and writing, a value that is never written can be returned by a read.

Principle of the algorithm The principle that underlies Lamport's mutex algorithm is inspired by ticket machines encountered in some shops (e.g., bakeries, hence the algorithm's name): you click the ticket machine to obtain the next ticket and wait until your ticket number is called. The problem lies in the fact that the implementation of such a counter requires mutual exclusion. To solve this issue, L. Lamport used the following solution: when a process wants to enter the critical section is compute a new ticket number, by looking at the ticket numbers that currently want to enter the critical section, takes the maximum and adds 1.

Due to concurrency, it is possible for two or more processes that simultaneously compute their ticket numbers to obtain the same number. So, in order to be able to order all the requests to enter the critical section, processes use *stamps*, which are pairs $\langle x, i\rangle$ where x is a ticket number and i is the identity of the process that issued the request. Given any two pairs $\langle x, i\rangle$ and $\langle y, j\rangle$, a total order relation is easily defined as follows:

$$\langle x, i\rangle < \langle y, j\rangle \stackrel{def}{=} \big((x < y) \vee (x = y \ \wedge \ i < j)\big).$$

Let us note that the process identities are used to break symmetry when two ticket numbers are equal. Moreover, the ticket values are assumed to be unbounded.

Shared non-atomic read/write registers To implement the previous idea in the non-atomic-RW model, as in the previous algorithm, the shared memory is made up of two arrays of SWMR safe Boolean registers, denoted $FLAG[1..n]$ and $MYTURN[1..n]$ (let us note that there is no MWMR register.)

$FLAG[i]$ is initialized to `false` and then alternates from `false` to `true` and `true` to `false`. $FLAG[i] =$ `true` means that p_i is computing its ticket number. $MYTURN[i]$ is initialized to 0, and contains the current ticket number of p_i. It is reset to its initial value 0 when p_i exits the critical section.

```
operation acquire() is
(1)  FLAG[i] ← true;
(2)  MYTURN[i] ← max(MYTURN[1], ..., MYTURN[n]) + 1;
(3)  FLAG[i] ← false;
(4)  for each j ∈ {1, ..., n} \ {i} do
(5)     wait (FLAG[j] = false);
(6)     wait ((MYTURN[j] = 0) ∨ ⟨MYTURN[i], i⟩ < ⟨MYTURN[j], j⟩)
(7)  end for;
(8)  return().

operation release() is
(9)  MYTURN[i] ← 0; return().
```

Figure 2: Lamport's 1974 n-process mutex algorithm (code for p_i) [53]

Lamport's Bakery algorithm This algorithm is described in Fig. 2. The operation acquire() is made up of two parts: a doorway (in which there is no wait statement), and a waiting room.

- In the doorway (lines 1-3), p_i computes its ticket number. As the registers $MYTURN[1..n]$ are only safe, the value obtained by p_i from $MYTURN[j]$ can be arbitrary if p_j is concurrently writing it. As already said, when a process p_j reads `true` from $FLAG[i]$, it knows that p_i is computing its ticket number from its reading of $MYTURN[1..n]$.
- In the waiting room (lines 4-7), p_i competes with each other process p_j. It first waits until $FLAG[i] = \texttt{false}$, and then waits again until either p_j is not interested in the critical section ($MYTURN[j] = 0$), or the stamp of its request is smaller than the one of p_j, namely $\langle MYTURN[i], i\rangle < \langle MYTURN[j], j\rangle$. The statement wait($P$), where P is a predicate, is a busy-waiting: it is a macro for "**repeat** skip **until** P **end repeat**".

The code of release() is a simple reset of $MYTURN[i]$ to 0. Let us observe that (as for Dijkstra's algorithm) due to the fact that process identities are used to address array entries, this algorithm is not symmetric.

Proof of the algorithm The proof is based on the following two lemmas (whose proofs can be found in [53, 73, 82]). These lemmas capture the essence of the algorithm. Let us say that a process p_i "is in the bakery" when it is executing any of the lines 4–9, i.e., when it is in the waiting room, inside the critical section, or executing release(). This algorithm satisfies the starvation-freedom property.

Lemma 1. *Let p_i and p_j be two processes that are in the bakery, and such that p_i entered the bakery before p_j enters the doorway. Then $MYTURN[i] < MYTURN[j]$.*

Lemma 2. *Let p_i and p_j be two processes such that p_i is inside the critical section while p_j is inside the bakery. Then $\langle MYTURN[i], i\rangle < \langle MYTURN[j], j\rangle$.*

The delicate point in proving the previous lemmas lies in the absence of atomicity. The reasoning has to be on the times at which processes enter and leave the doorway, the waiting room, and the bakery [59].

Bounding the domain of ticket numbers An algorithm, a variant of Lamport's Bakery algorithm, is presented in [81], which, assuming the registers are atomic (instead of safe), ensures that their size is bounded. Other algorithms based on bounded safe registers can be found in [73, 82].

The impact of Lamport's Bakery algorithm The design principles of this algorithm have had a *very strong impact on the understanding of asynchronous message-passing systems*. Among these impacts, there is the fact it uses SWMR registers. Such registers are easy to implement with a master copy residing at the writer process, and updated copies at all other processes. A second is the introduction of stamps which allows causality to be tracked despite asynchrony and the geographical distribution of computing entities [55]. More developments on this can be found in [62]. On another side, the algorithmics of safe registers combined with the techniques of "concurrent reading while writing" introduced in [54] can be seen as a multi-dimension pioneering work, a branch of which culminated in wait-free computing [44].

5 1981: Looking for Simplicity and Efficiency

A very simple two-process mutex algorithm In 1981 G. L. Peterson introduced a very simple two-process mutex algorithm for the atomic-RW communication model [68]. And simplicity is a first-class property [2, 34]!

This algorithm uses two SWMR atomic bits $FLAG[1..2]$, initialized to `false`, whose meaning is the same as in Dijkstra's algorithm, and an MWMR atomic register denoted $AFTERYOU$ whose initial value is irrelevant. It is described in Fig. 3.

```
operation acquire() is
(1) FLAG[i] ← true;
(2) AFTERYOU ← i;
(3) wait ((FLAG[j] = false) ∨ (AFTERYOU ≠ i));
(4) return().

operation release() is
(5) FLAG[i] ← false; return().
```

Figure 3: Peterson's 1981 two-process mutex algorithm (code for p_i) [68]

When a process p_i invokes acquire(), it raises its flag $FLAG[i]$ to inform the other process p_j it starts competing, and gives it the priority by writing its own identity in $AFTERYOU$. If the other process does not want to enter the critical section or has given p_i the priority, p_i enters the critical section. Otherwise, p_i waits until one of the previous conditions becomes true. In a very interesting way, this very simple algorithm ensures not only the mutex safety property but also satisfies the starvation-freedom property.

Let us suppress the SWMR registers $FLAG[1..2]$ from the algorithm. It is easy to see that the resulting algorithm satisfies the mutex safety property, but does not satisfy deadlock-freedom. We have the same result if, instead of suppressing the registers $FLAG[1..2]$, we suppress the atomic MWMR register $AFTERYOU$. Said another way, the composition of these two very simple algorithms, none of them guaranteeing deadlock-freedom, produces a starvation-free mutex algorithm.

An invariant-based proof of a variant of this algorithm is presented in [50]. In this variant, the MWMR atomic register $AFTERYOU$ is replaced by two SWMR atomic registers.

From two to n processes As shown in Fig. 4, Peterson's two-process algorithm can be generalized in a simple way to obtain an n-process starvation-free mutex algorithm.

Starting at level 0, the competing processes have to climb a ladder with $(n-1)$ levels. In the two-process algorithm, a process p_i uses a simple SWMR flag $FLAG[i]$ whose value is either `false` (to indicate it is not interested in the critical section) or `true` (to indicate it is interested). Instead of this binary flag, a process p_i uses now a multi-valued SWMR flag that progresses from a flag level to the next one. This flag, denoted $FLAGLEVEL[i]$, is initialized to 0 (indicating that p_i is not interested in the critical section). Process p_i increases first $FLAGLEVEL[i]$ to 1, then to 2, etc., until it attains the

```
operation acquire() is
(1)  for ℓ from 1 to (n − 1) do
(2)      FLAGLEVEL[i] ← ℓ;
(3)      AFTERYOU[ℓ] ← i;
(4)      wait ((∀ k ≠ i : FLAGLEVEL[k] < ℓ) ∨ (AFTERYOU[ℓ] ≠ i))
(5)  end for;
(6)  return().

operation release() is
(7)  FLAGLEVEL[i] ← 0; return().
```

Figure 4: Peterson's 1981 n-process mutex algorithm (code for p_i) [68]

level $(n-1)$ of the ladder. For $1 \leq x < n-1$, $FLAGLEVEL[i] = x$ means that p_i is at the level x of the ladder and, if $x < n-1$, it is trying to enter level $(x+1)$.

Moreover, to eliminate possible deadlocks at any level ℓ, $0 < \ell < n-1$, the processes use a second array of atomic registers $AFTERYOU[1..(n-1)]$ such that $AFTERYOU[\ell]$ keeps track of the last process that has entered level ℓ.

More precisely, a process p_i executes a for loop to progress from one level to the next one, starting from level 1 and finishing at level $n-1$. At each level the two-process solution is used to ensure that at most x processes are at level $(n-x)$. The predicate that allows a process to progress from level ℓ, $0 < \ell < n-1$, to level $\ell+1$ is similar to the one of the two-process algorithm. More precisely, p_i is allowed to progress to level $(\ell+1)$ if, from its point of view,

- either all the other processes are at a lower level (i.e., $\forall\, k \neq i :\ FLAGLEVEL[k] < \ell$),
- or it is not the last one that entered level ℓ (i.e., $AFTERYOU[\ell] \neq i$).

From mutex to k-mutex The k-mutex problem is a simple generalization of mutex, which allows up to k processes to be simultaneously in the critical section. Hence, mutex corresponds to the case $k = 1$. An extremely simple modification of Peterson's mutex algorithm generalizes it k-mutex. It consists in replacing at line 1 the loop upper bound $(n-1)$ by $(n-k)$, which suppresses the k higher levels of the ladder.

Exploiting the ladder idea The ladder idea has been used in fault-tolerant distributed computing to allow a process to obtain a view of which processes are concurrent with it. An example of such an use is the construction of an immediate snapshot object introduced in [16].

Looking for efficiency In each of the previous algorithms, an invocation of acquire() scans one or two arrays of size n (the number of processes), which means that its time complexity (measured by the number of processes to the shared memory) is at least $O(n)$. Hence, a natural question: is it possible to have a smaller time complexity?

As shown by G.L. Peterson and M.J. Fisher in [69] (1977), it is possible to use a tournament strategy to reduce the number of shared memory accesses. Kessels' algorithm [50] is based on this strategy, namely it uses a binary tree of instances of an underlying two-process mutex algorithm (namely, an adaptation of Peterson's two-process algorithm that, very interestingly, uses SWMR (single writer/multi-reader) shared variables only). Thanks to this tree, the gain obtained is from $O(n)$ to $O(\log_2 n)$. Such a mutex binary tree is depicted in Fig. 5, where (to simplify and without loss of generality) $n = 2^k$ with $k = 3$, i.e., there are 8 processes. Experimental results, which demonstrate the efficiency of several tournament algorithms, are presented in [19].

The tree is embedded in an array $LOCK[1..(n-1)]$ where $LOCK[1]$ is the root of the tree, and each non-leaf vertex $LOCK[x]$ has two children $LOCK[2x]$ and $LOCK[2x+1]$. According to its

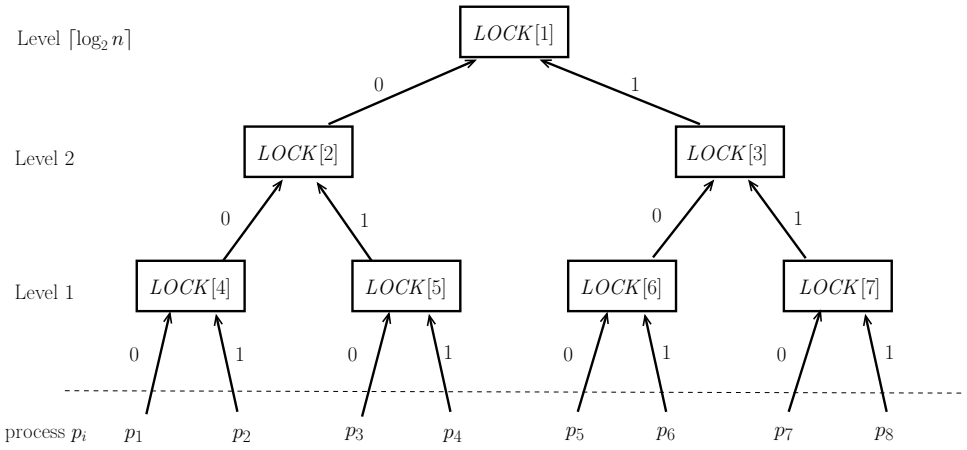

Figure 5: Tournament tree: from binary to n-process mutex

identity i, a process p_i start competing at a leaf vertex with a single other process p_j as defined by the tree. If it wins, it proceeds to the next parent vertex with the identity 0 or 1 as indicated in the figure. Hence, to enter the critical section, a process competes with $\log_2 n$ processes. Details of this algorithm are described in [50]. Pedagogical presentations and proofs are presented in [19, 73, 82] ([19] presents also experimental results).

6 1985: Using Real-Time

A synchrony assumption Differently from the previous algorithms that are asynchronous, the algorithm relies on a real-time assumption, i.e., it assumes that there a duration bound Δ, known by the processes, such that any two consecutive accesses (not separated by a delay, see below) to read/write registers by a process are separated by at most Δ time units. Let us note that, as Δ is known by the processes, it can be explicitly used in the algorithm.

Moreover, the system provides the processes with an operation $\mathsf{delay}()$ such that the invocation of $\mathsf{delay}(d)$, where d is a positive integer) stops the invoking process for a duration of at least d time units.

Fischer's algorithm The algorithm presented in Fig. 6 is due to M. J. Fischer. Actually, this algorithm has never been published by its author! It has its root in [58] and adapts the notion of a splitter (defined in the next section) to a real-time context. Section 3.4 of [1] gives more details on its history. Formal descriptions and proofs of it appear in [1, 24].

This simple, concise, and elegant algorithm needs a single atomic read/write register X, and works for any bounded number of processes n. Moreover, no process needs to know the number of processes.

The atomic MWMR register X is initialized to the default value $\perp$, and then contains the identity of a process or $\perp$. As in the previous algorithms, it is assumed that that the processes have distinct identities. Let us note that this algorithm is a *symmetric with equality* mutex algorithm.

When a process invokes $\mathsf{acquire}()$ it first waits until it sees $X = \perp$, namely, no process is inside the critical section. Then it competes with the other processes by writing its identity i into X, and stops during Δ units of times, after which it reads again X. If its identity is still in X, p_i terminates its invocation of $\mathsf{acquire}()$, otherwise, it re-enter the repeat loop. The code of $\mathsf{release}()$ is a simple reset of X to its initial value $\perp$.

Let us observe that, as the number of processes is bounded, in the presence of concurrent invocations of $\mathsf{acquire}()$, the last process that writes its identity in X will be granted the critical section. Hence, the algorithm is deadlock-free. Proofs of this algorithm can be found in [24, 73, 82].

```
operation acquire() is
(1) repeat wait(X = ⊥);
(2)        X ← i;
(3)        delay(Δ)
(4) until (X = i) end repeat;
(5) return().

operation release() is
(6) X ← ⊥; return().
```

Figure 6: Fischer's 1985 Δ-based n-process mutex algorithm (code for p_i) [36]

A fast timing-based algorithm is presented in [6, 7], where only five accesses to the shared memory are needed in order to enter a critical section in the absence of contention (and without the need to execute any delay statement). In the presence of contention, the winning process may need to delay itself for $2 \cdot \Delta$ time units. Chapter 10 of [82], covers timing-based mutex algorithms, including the case studied in [5] where the duration bound Δ exists but is unknown by the processes.

7 1987: Looking for Adaptivity

Reducing the cost of the operation acquire() **and** release() **in favorable circumstances** Considering the atomic-RW communication model, let us define the cost of an acquire() or release() operation as the number of read and write accesses to shared registers entailed by this operation.

As far as the operation acquire() is concerned, due to the busy-waiting (implemented by waiting loops) it is impossible to evaluate the operation's cost when several processes are competing. So, let us consider a favorable case (which occurs very often in practice), namely, when a process invokes acquire(), no process is inside the critical section and no other process is concurrently invoking acquire().

When considering the previous algorithms, as a process p_i does not know if it is or not in the previous favorable context, it must a priori consider it is competing with all the other processes and it has consequently to access all the shared registers to build a local view of the current global state. It is easy to see that the cost of the operation acquire() when a process runs alone (i.e., when there is no contention) of Dijkstra's algorithm and Lamport's algorithm is $O(n)$, while (due to the climbing of the ladder) the cost of Peterson's n-process mutex is $O(n^2)$.

A simple idea to reduce this cost consists in using a tournament tree, in which, at any time, a process competes with at most one other process; hence each node of the tree is a two-process mutex algorithm. The height of such a binary tree is $\lceil \log_2(n) \rceil$. It follows that in favorable circumstances, due to the binary tree traversal, the cost of acquire() and release() reduce to $O(\log_2(n))$ (see [73, 82] for more details). Hence, the question: Is it possible to design a deadlock-free n-process mutex algorithm the cost of which is constant for the operation acquire() in favorable circumstances, and the cost of release() is always constant? This question was posed and positively answered in 1987 by L. Lamport [58].

A basic skeleton: the splitter object This object provides the processes with a single operation denoted splitting(), that a process invokes at most once.

An invocation of splitting() returns a value from the set {`winner`, `lateloser`, `concurrentloser`}. When accessed by x processes, the object satisfies the following properties:

- At most $(x-1)$ processes obtain the value `concurrentloser`.
- At most $(x-1)$ processes obtain the value `lateloser`.
- At most one process obtains the value `winner`.

The algorithm described in Fig. 7 implements a splitter object on top of two MWMR atomic registers X, the initial value of which is irrelevant, and Y initialized to the default value $\perp$. Let us first note that the register X plays the same role in the splitter algorithm and in Fischer's synchrony-based algorithm described in Fig. 6. The main difference lies in the fact that the splitter algorithm considers a weaker asynchronous system. Hence, as Fischer's algorithm, Fig. 7 describes a symmetric with equality algorithm.

```
operation splitting() is
(1)  X ← i;
(2)  if (Y ≠ ⊥)
(3)     then return(lateloser)
(4)     else Y ← i;
(5)          if (X = i)
(6)            then return(winner)
(7)            else return(concurrentloser)
(8)          end if
(9)  end if.
```

Figure 7: Splitter object algorithm (code for p_i) [58]

It is easy to see that if a single process invokes splitting(), it returns `winner`. If several processes concurrently invoke splitting(), it is possible that all of them read $\perp$ from Y. Then, when they execute the lines 5-8, the last of them that wrote its identity in X will return `winner` at line 6, and all other will return `concurrentloser` at line 7. All the processes that will later invoke splitting() will read a non-$\perp$ value from Y and will consequently obtain the value `lateloser` at line 3. A proof of this algorithm can be found in [73].

Such a splitter constitutes the skeleton on which Lamport's *fast* mutex algorithm is based. In addition to the MWMR registers X and Y, this algorithm uses the same SWMR Boolean $FLAG[1..n]$ as in the previous algorithms. Hence, the algorithm building the operation acquire() in Fig. 8 is the algorithm of the splitter algorithm described in Fig. 7 enriched with appropriate statements. When looking at these two algorithms, the lines with the same number contain the very same statements. Line N3 replaces line 3, and the lines N7.1-N7.5 replace line 7. The lines N0, N1, and N10 are new lines.

```
operation acquire() is
(N0) repeat
(N1)   FLAG[i] ← true;
(1)    X ← i;
(2)    if (Y ≠ ⊥)
(R3)      then FLAG[i] ← false; wait (Y = ⊥)
(4)       else Y ← i;
(5)            if (X = i)
(6)              then return()
(R7.1)           else FLAG[i] ← false;
(R7.2)                wait (∀ j : 1 ≤ j ≤ n : FLAG[j] = false);
(R7.3)                if (Y = i) then return()
(R7.4)                           else wait (Y = ⊥)
(R7.5)                end if
(8)            end if
(9)    end if
(N10) end repeat.

operation release() is
(11)   Y ← ⊥; FLAG[i] ← false; return().
```

Figure 8: Lamport's 1987 n-process fast mutex algorithm (code for p_i) [58]

Lamport's Fast mutex algorithm This algorithm is described in Fig. 8. When a process invokes acquire() it enters a repeat loop that it exits when it will execute return() at line 6 or line R7.3.

Let us first consider the case where a process p_i invokes acquire() while no process is inside the critical section and no other process invokes acquire(). It is easy to see that, as Y is initialized to $\perp$, p_i terminates its invocation at line 6. Moreover, when it releases the critical section, it will reset its flag to `false` and Y to its initial value $\perp$ (line 11).

Let us now consider the case where a process p_i invokes acquire() and finds $Y \neq \perp$ when it executes line 2. Thanks to the splitter properties, we known that p_i is a lateloser, which means that a process is inside the critical section or other processes have invoked acquire() and are competing to enter the critical section. So, p_i momentarily withdraws itself from the competition. To this end, p_i first waits until it can have a chance to enter the critical section, namely until $Y = \perp$ (line R3). When this occurs p_i restarts to compete again by re-entering the repeat loop. Deadlock prevention is ensured by the wait statement wait $(Y = \perp)$ at line R3 and line R7.4 where p_i detects a conflict with respect to processes that progressed more "quickly" than it.

Let us now consider the case where a process p_i invokes acquire() and finds $Y = \perp$ when it executes line 2. So, p_i enters the code at lines 4-8 where, thanks to the splitter properties, we know that it is either a winner or an concurrentloser. Hence, from line 5), there are two cases.

- If $X = i$, p_i is the winner among the competing processes and it terminates its invocation of acquire() at line 6.
- If $X \neq i$, p_i is an concurrentloser, and consequently executes the lines R7.1-R7.5 at which (as before) it momentarily stops competing. This resignation period must not allow a deadlock to occur (maybe no other process has entered the critical section and other concurrentloser processes are still competing to win). To this end, p_i lowers it flag (line R7.1) and waits until it sees all flags down (let us remind that the readings of the flags are asynchronous). When the waiting period of line R7.2 stops, p_i reads Y, and there are two sub-cases.
  - If $Y \neq i$, another process wrote Y. In this case, p_i does at line 7.1 and line 7.4 the same as what is done (by lateloser processes) at line 3, and then re-enters the repeat loop.
  - If $Y = i$, no process executed line 4 after p_i wrote its identity into Y. It follows from the splitter read/write access pattern associated with the register Y (which is is the same as the one on the register X) that, as $p_i$is the last process that executed line 4 and, since this write, it has seen all the flags down, it can safely enter the critical section. Hence it terminates its invocation of acquire() (line R7.3).

Proofs of this algorithm can be found in [58, 73, 82].

Abortable mutex An abortable mutex algorithm is a mutex algorithm in which a process can decide to stop the execution of the operation acquire(). In this case, acquire() returns the default value $\perp$ (which does not allow it to enter the critical section). It is easy to see that Lamport's fast mutex algorithm can be made abortable by replacing only one or both its waiting statements (line R3 and line R7.4) by the statement return($\perp$) without altering the correctness of the algorithm.

Fast path vs. slow path This algorithm introduced the notion of *fast* path and *slow* path. Here the fast path of the acquire() operation is the sequence composed of the lines N0, N1, 1, 2, 4-6, which involves a constant number of accesses to atomic registers, namely 5. A slow path is any path taken by a process that is not a fast path. The operation release() involves two accesses to shared registers.

It follows that, in favorable circumstances (operation acquire() executed in a concurrency-free pattern), Lamport's algorithm requires 7 shared memory accesses for an instance of mutex.

The interested reader will find in [46] an efficient starvation-free fast path mutex algorithm that requires eight write and four read operations to enter and leave the critical section, which is claimed to be optimal for starvation-freedom.

Adaptive algorithms The notion of a fast algorithm has given rise to the more general notion of an *adaptive* algorithm. Intuitively, such an algorithm adapts itself to the concurrency degree [12, 28, 66, 65]. Fast mutex is adaptive in the sense that it provides operations with a constant number of accesses to shared registers in the contention-free case. The question of whether there exists a (fully) adaptive mutual exclusion algorithm was first raised in [65], where an adaptive algorithm is presented for a given working system, which is useful provided process creation and deletions are rare. In [65], the term *contention sensitive* was used, but later the term *adaptive* became commonly used.

Adaptivity can also be defined with respect to the quality of the result. As an example, let us consider the renaming problem in an n-process system in which any number of processes may crash [11]. Considering that the n processes have initial names belonging to a very large name space, the problem consists in designing an asynchronous crash-tolerant algorithm that assigns new names to the processes from a new name space as small as possible. It has been shown that $M = 2n - 1$ is a lower bound on the size of the new name space if the n processes participate in the renaming, except for some prime number-related values of n for which renaming can be implemented with $M = 2n - 2$ [25]. Size adaptive algorithms have been designed such that, if only p, $1 \leq p \leq n$, processes participate in the renaming, the size of the new name space can be reduced to $2p - 1$. Examples of such algorithms are described in [26].

8 1990: Local-Spinning Queue-Based Mutex

Taking into account the underlying shared memory architecture To obtain efficient executions, many multi-processors provide each processor with local cache memory. Such a local cache contains copies of shared registers composing the main memory. As a result, if a process p_i has an uptodate copy of a shared register R in its local cache, the cost for p_i to read R is (nearly) the same as the cost of accessing its local memory, i.e., no cost when compared to an access to the main memory. Differently, the write in a register R of a new value $v' \neq v$ (where v is the current value of R) by a process p_i entails an access to the main memory, which invalidates the value of R in the caches of the other processes. This memory architecture model is called *cache coherent* (CC) model. There is another architecture model called *distributed shared memory* (DSM) model. In this case, each processor manages parts of the shared memory, and there is no more a physical "main memory". These three shared memory models are illustrated in Fig. 9.

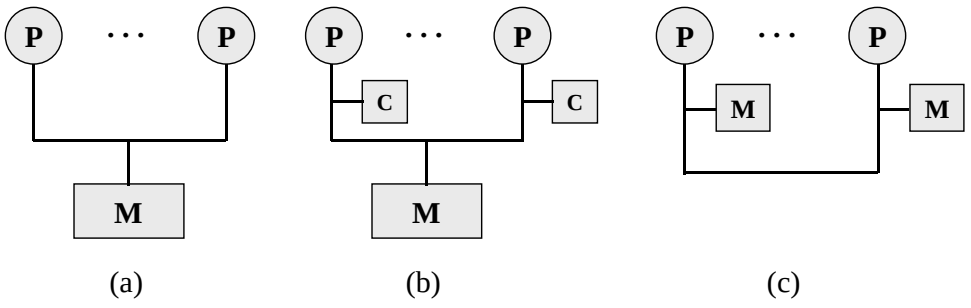

Figure 9: Three shared memory models. (a) Central shared memory. (b) Cache Coherent (CC). (c) Distributed Shared Memory (DSM). P denotes processor, C denotes cache, M denotes shared memory

Local spinning Hence, the question: "Is it possible to benefit from cache memories in order to decrease the cost of the operations acquire() and release()?" It turns out that the answer to this question is positive as shown below. This is due to the fact that the waiting loops accessing shared registers that have their last values in local caches become local waiting loops. This, called *local spinning*, decreases the underlying network contention.

8.1 Local spinning in the CC model

Among the many local-spinning mutex algorithms for the CC model, each with its specific properties, due to its simplicity, the one presented here is due to T. E. Anderson [10]. In addition to the operations read and write, this algorithm uses the fetch&increment operation (which it applies to a single predefined shared register). Hence, the algorithm assumes the atomic-RW+ communication model.

Shared registers and local variables A process p_i manages a single local variable denoted $ticket_i$. The shared memory is made up of the following registers.

- $TICKET$, initialized to 0, is the only register accessed with the fetch&increment operation. It is used by a process to obtain a sequence number that will allow it to enter the critical section. It is assumed that $TICKET$ can increase forever (an algorithm where $TICKET$ is bounded is presented in [9]).
- $VALID[0..(n-1)]$ is an array of n atomic MWMR bits. $VALID[0]$ is initialized to 1, while all its other entries are initialized to 0.

Local-spinning queue-based mutex Anderson's algorithm is described in Fig. 10. It is queue-based in the sense that it builds a "queue of requests" ordered by their sequence numbers.

Let us first observe that, due to the fetch&increment() operation invoked at line 1, all the invocations of acquire() obtains different increasing integer values. To make the understating easier, let us assume that the array $VALID$ has an infinite number of entries. The idea is

- on the safety side: maintain invariant the following predicate $\Sigma_{x\geq 0}\, VALID[x] \leq 1$, and
- on the liveness side: allow a process p_i to progress (starvation-freedom) when its ticket $ticket_i$ is such that $VALID[ticket_i] = 1$ (line 2).

The invariant $\Sigma_{x\geq 0}\, VALID[x] \leq 1$ is initially true, and then maintained true by the process that invokes release() when it executes line 4 (at which it announces that it exits the critical section) and line 5 (at which it allows the process whose ticket is the next sequence number to enter the critical section).

```
operation acquire() is
(1)  ticket_i ← fetch&increment(TICKET);
(2)  wait(VALID[ticket_i mod n] = 1);
(3)  return().

operation release() is
(4)  VALID[ticket_i mod n] ← 0;
(5)  VALID[(ticket_i + 1) mod n] ← 1;
(6)  return().
```

Figure 10: Anderson's 1990 local-spinning n-process mutex algorithm (code for p_i) [10]

Let us observe that, while the values obtained from $TICKET$ increase forever, each ticket value is associated with exactly one process, and the next invocation of acquire() by a process is possible only after it invoked the operation release() associated with its previous invocation of acquire(). In short, a process is engaged in at most one call of acquire() at a time. It follows that, starting from the entry

$ticket_i$, such that $VALID[ticket_i] = 1$, only the entries $VALID[ticket_i + 1]$, ..., $VALID[ticket_i + (n - 1)]$, are meaningful (in the worst case each other process acquired a ticket value). From this observation and the fact that the sequence of possible ticket values is the sequence made up of the consecutive positive integers, we conclude that, instead of being an array with an infinite number of entries, $VALID$ can be folded in a cyclic array with n entries only, which is unambiguously obtained from the modulo n function. Let us finally note that no two processes spin on the same register at the same time.

Properties of the algorithm The operation acquire() by a process p_i always costs one access to main shared memory (access to $TICKET$), plus one access to $VALID[ticket_i \bmod n]$ when the copy in its local cache has been invalidated. The operation release() costs two accesses to the main shared memory. So the complexity, measured as the number of accesses to the main shared memory, is constant for each use of the critical section.

Moreover, as it does not use the identities of the processes, Anderson's algorithm works in a process-anonymous system, and consequently it trivially satisfies the full symmetry property. The initial symmetry (same code, same initialization of the local variables) is broken thanks to the use of the operation fetch&increment which allows the processes to create an asymmetry that is sufficient to solve mutex).

8.2 Local spinning in both the CC model and the DSM model

Anderson's algorithm is specific to the CC model in the sense that, if executed in the DSM model, it remains correct but loses its local-spinning property, namely its complexity with respect to the number of remote shared memory accesses becomes unbounded. In this section, we present a local-spinning algorithm, due to J.M. Mellor-Crummey and M.L. Scott [64], which is described in Fig. 11. The algorithm works in both models and whose complexity of the acquire() and release() operations is constant in both architecture models.

Extended atomic-RW+ communication model This operation model is the model atomic-RW+ introduced in Section 2 in which the operation fetch&increment is replaced by a swap operation defined as follows:

- The invocation of swap(R, aux_i) by a process p_i atomically exchanges the value in the shared register R and the value in p_i's local variable aux_i.

The operations that a process can invoke are consequently (a) read and write on all shared registers on all registers except one, and (b) swap() and compare&swap() on the remaining register. As some variables contain pointers, we use the following notations. Let X denote a register containing a pointer value and Y denote a register containing a non-pointer value.

- $\downarrow X$ denotes the register point to by X.
- $\uparrow Y$ denotes a pointer to Y.
- Hence $\downarrow (\uparrow Y)$ is the non-pointer register Y and $\uparrow (\downarrow X)$ is the pointer register X.

Local variables and shared registers Let an "element" be a record composed of two fields: a bit denoted $value$ and a pointer denoted $next$. The shared memory is composed of $(n + 1)$ registers, namely an array $NODE[0..(n - 1)]$ containing elements, and a register $TAIL$ which contains a pointer initialized to the null pointer denoted $\perp$.[3] $TAIL$ is aimed at pointing to the last element of a FIFO queue. Thanks to the pointer field in each element, the algorithm will manage elements in $NODE[0..(n-1)]$ so that they will compose a queue including all the pending requests issued by the processes, the element at

[3]The algorithm described [64] does not require an array bounded by n (the number of processes). We consider a bounded array to simplify the presentation.

the head of the queue being the process that is in the critical section, and its last element being pointed to by $TAIL$.

The local memory of a process p_i contains three local variables which are pointers to elements in the shared memory. The local variable $mynode_i$ is initialized to $\uparrow NODE[i]$ and forever keeps this value. The local variables $previous_i$ and $successor_i$ are modified by p_i but do not need to be initialized.

```
operation acquire() is
(1)  (↓ mynode_i).next ← ⊥;
(2)  previous_i ← mynode_i;
(3)  swap(TAIL, previous_i);
(4)  if previous_i ≠ ⊥
(5)     then (↓ mynode_i).value ← 0;
(6)          (↓ previous_i).next ← mynode_i;
(7)          wait((↓ mynode_i).value = 1)
(8)  end if;
(9)  return().

operation release() is
(10) if (↓ mynode_i).next ≠ ⊥
(11)    then successor_i ← (↓ mynode_i).next;
(12)         (↓ successor_i).value ← 1
(13)    else if ¬ compare&swap(TAIL, mynode_i, ⊥)
(14)            then wait((↓ mynode_i).next ≠ ⊥);
(15)                 successor_i ← (↓ mynode_i).next;
(16)                 (↓ successor_i).value ← 1
(17)         end if
(18) if;
(19) return().
```

Figure 11: Mellor-Crummey & Scott's 1991 local-spinning n-process mutex algorithm (code for p_i) [64]

Algorithm of the operation acquire() When a process p_i invokes acquire() it builds a new element pointed to by $mynode_i$ (line 1), copies this pointer in $previous_i$ (line 2) and adds its request at the tail of the queue with the help of the atomic swap operation (line 3). Hence, the content of $previous_i$ becomes the end of the queue and the previous last element is saved in $previous_i$. When this is done, it checks the previous last element of the queue, which, as mentioned, is now in $previous_i$ (line 4). There are two cases.

- If there no such element ($previous_i = \bot$), p_i terminates its invocation of acquire().
- If $previous_i \neq \bot$), p_i assigns 0 to $\downarrow mynode_i$ to indicate it is not yet at the head of the queue (line 5), places this information in the queue (line 6), and waits until $(\downarrow mynode_i).value = 1$, which is the signal indicating it is now at the head of the queue. When this will happen, p_i will terminate its invocation of acquire() allowing it to enter the critical section (line 7).

Algorithm of the operation release() When a process p_i invokes release(), it checks if it has a successor in the queue (predicate $(\downarrow mynode_i).next \neq \bot$ at line 10). There are two cases.

- If $(\downarrow mynode_i).next \neq \bot$, p_i sets to 1 the value field of its successor (lines 11-12), which allows its successor in the queue to stop waiting at line 7, and consequently terminate its invocation of acquire().
- If $(\downarrow mynode_i).next = \bot$, p_i checks if $TAIL = mynode_i$ with the atomic compare&swap operation (line 13).
  - If the operation returns `true`, p_i was alone in the queue. In this case, the operation release() terminates.

– If the operation returns `false`, $p_i$has a successor, but due to asynchrony, this is not yet registered in $(\downarrow mynode_i).next$. So, p_i waits until this registration is done (line 14). When this occurs p_i allows its successor to progress (line 15 by executing the same statements as at lines 11-12).

Properties of Mellor-Crummey and Scott's algorithm Considering a atomic-RW model extended with the swap and compare&swap operations, the algorithm satisfies the following properties.

- It implements starvation-free mutex.
- A process locally spins on a single register ($mynode_i$) and no two processes locally spin on the same shared register.
- It satisfies local spinning in both the CC and the DSM architecture model, from which follows that the time complexity of both operations are constant.
- It "nearly" satisfies the full symmetry property. The process identities are used only in the initialization part, namely, a process p_i uses its identity only to initialize $mynode_i$ to a pointer to $NODE[i]$. Actually, the algorithm needs to associate one and only one entry of $NODE[1.n]$ to each process. Hence,if we assume that this pairing is initially done by a "magical" function that associates a shared register with each process p_i, the algorithm satisfies the full symmetry property.
- This algorithm does not satisfy the important *bounded exit* property (which is satisfied by all the other algorithms presented in this article). This property states that the operation release() involves a fixed number of steps of the invoking process (in other words, this operation does not contain a wait statement). There are variants of this algorithm that satisfy the bounded exit property while still meeting the properties mentioned above [35].
- A proof of this algorithm can be found in [47].

Other efficient local spinning algorithms can be found in [8, 35, 82].

9 2020: Mutex Despite Full Anonymity

Process anonymity In a process-anonymous system, it is not possible to distinguish a process from another process. So, not only do the processes have no name, but they have the same code with the same initialization of their local variables.

Process anonymity in shared memory systems has been studied in several articles mainly from a computability point of view or a fault-tolerance point of view, e.g., [13, 17, 41] to cite a few. A survey on process anonymity in read/write systems is presented in [75].

Memory anonymity Memory anonymity has been introduced a few years ago (2017) in [83]. Let us see the shared memory as an array of m registers: $R[1..m]$. In the non-anonymous memory model, for any x, $R[x]$ denotes the same register for all the processes. This is no longer true in an anonymous memory: $R[x]$ invoked by p_i and $R[x]$ invoked by p_j do not necessarily denote the same register. More precisely, a memory-anonymous system is such that:

- For each process p_i an adversary defined a permutation $f_i()$ over the set $\{1, 2, \cdots, m\}$, such that when p_i uses the address $R[x]$, it actually accesses $R[f_i(x)]$,
- No process knows the permutations, and
- All the registers are initialized to the same default value denoted $\perp$.

Let us observe that, due to their anonymity, the access to an anonymous register cannot be restricted to a given process, i.e., any process can access any anonymous register. Hence all the registers are MWMR.

An example of anonymous memory made up of three registers is presented in Table 2. To emphasize the fact that $R[x]$ can have a different meaning for different processes, we write $R_i[x]$ when p_i invokes $R[x]$.

identifiers for an external observer	local identifiers for process p_i	local identifiers for process p_j
$R[1]$	$R_i[2]$	$R_j[3]$
$R[2]$	$R_i[3]$	$R_j[1]$
$R[3]$	$R_i[1]$	$R_j[2]$
permutation	$f_i() : [2, 3, 1]$	$f_j() : [3, 1, 2]$

Table 2: Illustration of the anonymous memory model

Problems such as leader election, agreement, mutual exclusion, renaming, memory de-anonymization have been addressed in memory-anonymous systems where the processes are not anonymous [3, 38, 39, 76, 83]. Interestingly, anonymous shared memory systems are useful in modeling biologically inspired distributed computing methods, especially those that are based on ideas from molecular biology [67, 71].

Necessary and sufficient conditions As far as deadlock-free mutex is concerned in an n-process system where communication is through an anonymous memory of size m, and where process identities can only be compared (symmetry constraint), the following results are known. Let $M(n)$ be the set of all the integers relatively prime with 2, ..., n, i.e., $M(n) = \{m \text{ such that } \forall\, \ell : 1 < \ell \leq n : \mathsf{gcd}(\ell, m) = 1\}$.

- If the communication model is atomic-RW: deadlock-free mutex can be solved if and only if $m \in M(n) \setminus \{1\}$. The *only if* direction was proven in [83], the *if* direction was proven in [3].
- If the communication model is atomic-RW+ where the additional operation is $\mathsf{compare\&swap}()$: deadlock-free mutex can be solved if and only if $m \in M(n)$ [3].

These necessary and sufficient conditions provide the asymmetry seed needed to solve mutex in anonymous memory symmetric systems.

Mutex in fully anonymous systems A fully anonymous system is a system in which *both* the processes and the memory are anonymous. It is shown in [77] that deadlock-free mutex (a) cannot be solved if communication is through read/write registers, and (b) can be solved if communication is through read/write/compare&swap registers if and only if $m \in M(n)$.

As in a fully anonymous system, no process has an identity; any algorithm designed for such a system trivially satisfies the "symmetric with equality" property.

A fully anonymous read/write/compare&swap algorithm The algorithm presented in Fig. 12 is due to M. Raynal and G. Taubenfeld[4]. A detailed proof of it can be found in [78].

Let $R[1..m]$ denote the anonymous memory, where each register is initialized to 0. Each process p_i manages a local array of bits $myview_i[1..m]$, all initialized to `false`, and three local variables denoted $counter_i$, $round_i$, and $competitors_i$.

A process *owns* an anonymous register when the register has a positive value and the process is the last one that wrote this positive value. The local variable $counter_i$ stores the number of registers *owned* by p_i. The bit $myview_i[j]$ is true if and only if p_i owns the register $R_i[j]$.

[4]A previous version of this fully anonymous read/write/compare&swap mutex algorithm presented in [77] had a faulty scenario in which deadlock-freedom was not ensured.

```
operation acquire() is
(1)  round_i ← 0; counter_i ← 0; for each j ∈ {1, ..., m} do myview_i[j] ← false end for;
(2)  repeat
(3)    round_i ← round_i + 1;
(4)    for each j ∈ {1, ..., m} such that myview_i[j] do R_i[j] ← round_i end for;
(5)    for each j ∈ {1, ..., m} do
(6)      while R_i[j] < round_i do
(7)        myview_i[j] ← compare&swap(R_i[j], 0, round_i);
(8)        if myview_i[j] then counter_i ← counter_i + 1 end if
(9)      end while
(10)   end for;
(11)   competitors_i ← n − round_i + 1;
(12)   if counter_i × competitors_i < m then
(13)     round_i ← 0; counter_i ← 0;
(14)     for each j ∈ {1, ..., m} such that myview_i[j] do myview_i[j] ← false; R_i[j] ← 0 end for;
(15)     wait(∀ j ∈ {1, ..., m} : R_i[j] = 0) end if
(16) until round_i = n end repeat;
(17) return().

operation release() is
(18) for each j ∈ {1, ..., m} do R_i[j] ← 0 end for;
(19) return().
```

Figure 12: Deadlock-free mutex for n anonymous processes and an anonymous memory of size $m \in M(n)$

Algorithm of the acquire() **operation** A process p_i first initializes its local variables (line 1) and then enters a "repeat" loop, that it will exit when $round_i = n$ (this condition can be replaced by repeating until $myview_i = [\texttt{true}, ..., \texttt{true}]$). Its behavior inside this loop is composed of three parts.

- When it enters a new round $round_i = r$, a process p_i first writes its new round number in the registers it owned previously (line 4). Those registers are locally registered in the local array $myview_i[1..m]$ (which contains only the value `false` when p_i executes the first round).
- Then, p_i strives to own more registers by writing its current round number r in new registers (lines 5-10). To this end, p_i considers each register such $R_i[j] < round_i$ (lines 6-8), and for each of them, loops until $R_i[j] \geq round_i$. To solve possible conflicts in the gain for new registers, the processes use the atomic operation compare&swap().
- Finally, when at line 10, p_i exits the "for" loop it entered at line 5, it computes the maximal number of competing processes (line 11). If the number of registers it owns (registered in $counter_i$) is smaller than the average number of registers owned by the competing processes (line 12), p_i resigns by resetting to their initial values $counter_i$, $round_i$ (line 13) the entries of the array $myview_i[1..m]$ currently equal to `true` and the anonymous registers it owns (line 14). Then it waits until it sees all the registers equal to 0.

  If p_i does not resign (the withdrawal predicate at line 12 is false), it progresses to the next round if $rounds_i < n$ or enters the critical section if $rounds_i = n$ (line 18).

Algorithm of the release() **operation** When a process p_i invokes release(), it releases all the anonymous registers by writing 0 into all of them one at a time (line 18). Notice that a competing process (if any) may see some anonymous registers as being released (i.e., equal to 0) and other registers as being not yet released (i.e., still equal to n). This will cause no problem since in such a case, the process will either continue to the next round (if it owns enough registers) or release all the registers it owns and waits in line 15 until it sees each anonymous register equal to 0.

On the updates of $myview_i[j]$ It is easy to see that the assignment $myview_i[j] \leftarrow$ `true` executed by p_i (line 7) has $R_i[j] = 0$ as pre-condition and $R_i[j] = round_i > 0$ as post-condition. Similarly the assignment $myview_i[j] \leftarrow$ `false` (line 14) is executed with $R_i[j] = round_i > 0$ and followed by the statement $R_i[j] \leftarrow 0$ to obtain $R_i[j] = 0$ as post-condition. It follows that the sum of all the counters of the competing processes is equal to the number of registers that have positive values. Moreover, due to the compare&swap() operation, each register with a positive value is owned by exactly one process.

10 By Way of Conclusion

Considering process failures Mutex cannot be solved in read/write systems where processes may simply crash in its critical section (a crash is a premature definitive stop). Thus, some additional assumptions must be made to cope with process failures.

In Section 3.3 of [82], mutex algorithms are considered that are immune to some type of process failures, where a process may repeatedly fail and restart. Informally, a process fails by returning to the beginning of its code (also called the remainder code) and resetting all the registers for which it has write access to their predefined default values. This means that registers that may be written by more than one process cannot be used since one of the processes that has the write permission may fail and "take the shared register with it". Thus, in such a case, only SWMR registers may be used.

Another approach consists in enriching the model by providing the processes with information on failures. This is the failure detector-based approach introduced in [27]. The weakest information on failures to solve mutex has been established in [15, 30]. Assuming atomic registers, a crash-tolerant version of Lamport's Bakery mutex algorithm is described in [30].

An important recent advancement in hardware technology is developing non-volatile random-access memory (NVRAM), which retains its state despite a power outage. When a crash occurs, it causes the CPU registers' values (including the program counters) to be set to arbitrary values, but any pieces of the algorithm's state stored in the NVRAM remain intact. So, when a process p_i restarts after a crash, instead of redoing everything from the beginning, it might be possible for p_i to "recover" its state by consulting the NVRAM variables and resume the computation from where it was at the time of the crash. This new hardware technology has initiated research on the design of recoverable mutual exclusion algorithms. In [40], Golab and Ramaraju were the first to formalize the recoverable mutual exclusion problem and designed several algorithms that are resilient to process (or system) crashes by adapting traditional mutual exclusion algorithms.

A natural extension of mutex: group mutex This problem, introduced by Y. Joung in [48], is a generalization of the mutual exclusion and readers-writers problems [29]. It ensures mutual exclusion on different groups of processes in accessing a critical section (resource) while allowing processes of the same group to share the critical section (resource). More developments on group mutex can be found in [42, 43, 49, 63].

An operating system view The interested reader can consult [18] and [33] for an early history of concurrent programming. These articles are oriented towards the concept of a process, basic synchronization mechanisms, and operating system architecture.

Books on mutex A very first book entirely dedicated to mutex is [72], that considered algorithms published until 1986. A presentation of mutex algorithms published after 1986 appeared in [9]. Then was published [82], which is entirely devoted to synchronization. Several books have several full chapters devoted to mutex (e.g, [14, 37, 45, 52, 61, 73] to cite a few).

Mutex in message-passing systems This date-based history of mutex was devoted to shared memory systems. The reader will find a survey on mutex algorithms for asynchronous message-passing in Part III of [73] (more generally, this book is devoted to algorithms in failure-free asynchronous message-passing systems.) Among the very first message-passing mutex algorithms there are the ones described in [54, 60]. The reader will also find an example-based intuitive introduction to distributed computing in [84] and an introduction to self-stabilizing algorithms in [4], both including mutex algorithms.

From mutex to consensus Recently appeared in [70] a short history on evolution from mutex to consensus (i.e., the inescapable need for processes to agree in order to cooperate correctly in the presence of adversaries such as asynchrony and failures).

Acknowledgments

We thank all the referees for their constructive comments that helped us improve the article's presentation and technical content. A special thanks go to Peter A. Buhr and Wojciech Golab (University of Waterloo, Canada) who, thanks to an insightful analysis (based on testing and model checking tests), discovered a deadlock-prone scenario in the fully anonymous mutex algorithm. Interestingly enough, the corrected version presented in this article (that passed the testing and model checking tests) appears to be much simpler than the previous faulty version.

References

[1] Abadi M. and Lamport L., An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):543–1571 (1994)

[2] Aigner M. and Ziegler G., Proofs from THE BOOK (4th edition). Springer, 274 pages, ISBN 978-3-642-00856-6 (2010)

[3] Aghazadeh Z., Imbs D., Raynal M., Taubenfeld G., and Woelfel Ph., Optimal memory-anonymous symmetric deadlock-free mutual exclusion. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM Press, pp. 157-166 (2019)

[4] Altisen K., Devismes S., Duboid S., and Petit F., *Introduction to distributed self-stabilizing algorithms.* Synthesis Lectures on Distributed Computing Theory, Morgan Ćlaypool, 167 pages, ISBN 9781681735368 (2019)

[5] Alur R., Attiya H., and Taubenfeld G. Time-adaptive algorithms for synchronization. *SIAM Journal on Computing*, 26(2):539-56 (1997)

[6] Alur R. and Taubenfeld G., Results about fast mutual exclusion. *Proceedings of the 13th IEEE Real-Time Systems Symposium*, IEEE Press, pp. 12–21 (1992)

[7] Alur R. and Taubenfeld G., Fast timing-based algorithms. *Distributed Computing*, 10(1):1-10 (1996)

[8] Anderson J.H., and Kim Y.-J., Adaptive mutual exclusion with local spinning. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer LNCS 1914, pp. 29-43 (2000)

[9] Anderson J.H., Kim Y.-J., and Herman T., Shared memory mutual exclusion: major research trends since 1986, *Distributed Computing*, 16:75-110 (2003)

[10] Anderson T.E., The performance of spin lock alternatives for shared-memory multiprocesssor. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6-16 (1990)

[11] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548 (1990)

[12] Attiya H. and Fouren A., Algorithms adapting to contention point. *Journal of the ACM*, 50(4):444–468 (2003)

[13] Attiya H., Gorbach A., and Moran. S., Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183 (2002)

[14] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics, (2nd Edition)*, Wiley-Interscience, 414 pages, ISBN 0-471-45324-2 ,(2004)

[15] Bhatt V. and Jayanti P., On the existence of weakest failure detectors for mutual exclusion and k-exclusion. *23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 325-339 (2009)

[16] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51 (1993)

[17] Bouzid Z., Raynal M., and and Sutra P., Anonymous obstruction-free (n, k)-set agreement with $(n - k + 1)$ atomic read/write registers. *Distributed Computing*, 31(2):99-117 (2018)

[18] Brinch Hansen P. (Editor), *The origin of concurrent programming*. Springer, 534 pages (2002)

[19] Buhr P.A., Dice D., and Hesselink W.H., High-performance N-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience,* 27(3):651-701 (2015)

[20] Buhr P.A., Dice D., and Hesselink W.H., Dekker's mutual exclusion algorithm made RW-safe. *Concurrency and Computation: Practice and Experience,* 28(1):144-165 (2016)

[21] Burns J.E., Symmetry in systems of asynchronous processes. *Proc. 22d IEEE Symposium on Foundations of Computer Science*, IEEE Press, pp. 169-174 (1981)

[22] Burns J.E., Jackson P., Lynch N. A., Fisher M.J., and Peterson G.L., Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183-205 (1982)

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

[24] Carruth J.A. and Misra J., Proof of a real-time mutual exclusion algorithm. *Parallel Processing Letters*, 6(2):251-257 (1996)

[25] Castañeda A., Rajsbaum S., New combinatorial topology upper and lower bounds for renaming: the upper bound. *Journal of the ACM*, 59(1), Article 3 (2012)

[26] Castañeda A., Rajsbaum, and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251 (2011)

[27] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

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

[29] Courtois P.J., Heymans F., and Parnas D.L., Concurrent control with readers and writers. *Communications of the ACM*, 14(5):667-668 (1971)

[30] Delporte-Gallet C., Fauconnier H., and Raynal M., On the weakest failure detector for read/write-based mutual exclusion. *Proc. 33nd Int'l Conference on Advanced Information Networking and Applications (AINA'19)*. Springer AISC 926, pp. 272-285 (2019)

[31] Dijkstra E.W., Over de sequentialiteit van procesbeschrijvingen (on the nature of sequential processes). *EW Dijkstra Archive (EWD-35)*, Center for American History, University of Texas at Austin (Translation by Martien van der Burgt and Heather Lawrence) (1962)

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

[33] Dijkstra E.W., Cooperating sequential processes. In *Programming Languages (F. Genuys Ed.)*, Academic Press, pp. 43-112 (1968)

[34] Dijkstra E.W., Some beautiful arguments using mathematical induction. *Algorithmica*, 13(1):1-8 (1980)

[35] Dvir R. and Taubenfeld G., Mutual exclusion algorithms with constant RMR complexity and wait-free exit code. *Proc. 21st Int'l Conference on Principles of Distributed Systems (OPODIS'17)*, LIPICS Vol. 95, Article 17, 16 pages (2017)

[36] Fischer M., Re: Where are you? *E-mail message to Leslie Lamport*. Arpanet message sent on June 25, 1985 18:56:29 EDT, number 8506252257.AA07636@YALE-BULLDOG.YALE.ARPA, 47 lines (1985)

[37] Garg V.K., *Elements of Distributed Computing*. Wiley-Interscience, 423 pages (2002)

[38] Godard E., Imbs D., Raynal M., and Taubenfeld G., Leader-based de-anonymization of an anonymous read/write memory. *Theoretical Computer Science*, 836(10):110-123 (2020)

[39] Godard E., Imbs D., Raynal M., and Taubenfeld G., From Bezout identity to space-optimal leader election in anonymous memory systems. *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC'20)*, ACM press, pp. 41-50 (2020)

[40] Golab W. and Ramaraju A., Recoverable mutual exclusion (Extended abstract). *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 65–74 (2016)

[41] Guerraoui R. and Ruppert E., Anonymous and fault-tolerant shared-memory computations. *Distributed Computing*, 20:165-177 (2007)

[42] Hadzilacos V., A note on group mutual exclusion. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press, pp. 100-106 (2001)

[43] He Y., Gopalakrishnan K., and Gafni E., Group mutual exclusion in linear time and space. *Proc. 17th Conference on Distributed Computing and Networking (ICDCN'16)*, ACM Press, pp 22:1–22:10 (2016)

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

[45] Herlihy M. and Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, ISBN 978-0-12-370591-4 (2008)

[46] Hesselink W.H., Buhr P.A., and Dice D., Fast mutual exclusion by the trinagle algorithm. *Concurrency and Computation: Practice and Experience,* 30(4) (2018)

[47] Johnson T. and Harathi K., A simple correctness proof of the MCS contention-free lock. *Information Processing Letters*, 48(5);215-220 (1993)

[48] Joung Y., Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206 (2000)

[49] Keane P. and Moir M., A simple local-spin group mutual exclusion algorithm. *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, ACM Press, pp. 23-32 (1999)

[50] Kessels J.L.W., Arbitration without common modifiable variables. *Acta Informatica*, 17(2):135-141 (1982)

[51] Knuth D.E., Solution of a problem in concurrent programming control. *Communications of the ACM*, 9(5):321-322 (1966)

[52] Kshemkalyani A.D. and Singhal M., *Distributed computing: principles, algorithms and systems*. Cambridge University Press, 736 pages (2008)

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

[54] Lamport L., Concurrent reading while writing. *Communications of the ACM*, 20(11):806-811 (1977)

[55] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)

[56] Lamport L., On inter-process communications, part I: basic formalism. *Distributed Computing*, 1(2): 77-85 (1986)

[57] Lamport L., The mutual exclusion problem: Part I- a theory of interprocess communication. *Journal of the ACM*, 33(2): 313-326 (1986)

[58] Lamport L., Fast mutual exclusion. *ACM Transactions on Computer Systems*, 5(1):1-11 (1987)

[59] Lamport L., The computer science of concurrency: the early years (Turing lecture). *Communications of the ACM*, 58(6):71-76 (2015)

[60] Le Lann G., Distributed systems: towards a formal approach. *IFIP World Congress*, pp. 155–160 (1977)

[61] Lynch N. A., *Distributed algorithms*. Morgan Kaufmann Pub., 872 pages, ISBN 1-55860-384-4 (1996)

[62] Malkhi D., (Editor) *Concurrency: the works of Leslie Lamport.* Morgan & Clapypool, 333 pages, ISBN 978-1-4503-7270-1 (2019)

[63] Maor L. and Taubenfeld G., Constant RMR group mutual exclusion for arbitrarily many processes and sessions. *Proc. 35th Int'l Symposium on Distributed Computing (DISC'21)*, LIPICS Vol. 209, pp. 30:1-30:16 (2021)

[64] Mellor-Crummey J.M. and Scott M.L, Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9(1):21–65 (1991)

[65] Merritt M. and Taubenfeld G., Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993. (Published as an AT&T technical memorandum, May 1991.)

[66] Merritt M. and Taubenfeld G., Computing with infinitely many processes. *Information and Computation*, 233:12-31 (2003) *ACM Transactions on Computer Systems* 9(1):21–65 (1991)

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

[68] Peterson G.L., Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115-116 (1981)

[69] Peterson G.L. and Fischer M.J., Economical solutions for the critical section problem in a distributed system. *Proc. 9th ACM Sympoium on Theory of Computing (STOC'77)*, ACM Press, pp. 91-97 (1977)

[70] Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking: A half-century evolution. *Communications of the ACM*, Vol. 63(1):78-87 (2020)

[71] Rashid S., Taubenfeld G. and Bar-Joseph Z., The epigenetic consensus problem. *Proc. 28th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'21)*, Springer LNCS 12810, pp. 146-163 (2021)

[72] Raynal M., *Algorithms for mutual exclusion*. The MIT Press, 107 pages (1986) [Translation of a French version published in 1984]

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

[74] Raynal M., *Distributed algorithms for message-passing systems. Springer*, 510 pages, ISBN 978-3-642-38122-5 (2013)

[75] Raynal M. and Cao J., Anonymity in distributed read/write systems: a short introduction. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS 11028, pp. 122-140 (2018)

[76] Raynal M. and Taubenfeld G., Fully anonymous consensus and set agreement algorithms. *Proc. 8th Int'l Conference on Networked Systems (NETYS'20)*, Springer LNCS 12129, pp. 314-328 (2020)

[77] Raynal M. and Taubenfeld G., Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, Vol. 158, 105938, 7 pages (2020)

[78] Raynal M. and Taubenfeld G., Corrigendum: Mutual exclusion in fully anonymous shared memory systems. Sent to *Information Processing Letters* for correction of [77].

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

[80] Styer E. and and Peterson G.L., Tight bounds for shared memory symmetric mutual exclusion problems. *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pp. 177-191 (1989)

[81] Taubenfeld G., The black-white bakery algorithm. *Proc. 18th Int'l Symposium on Distributed Computing (DISC'04)*, Springer LNCS 3274, pp. 56-70 (2004)

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

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

[84] Taubenfeld T., *Distributed computing pearls.* Synthesis Lectures on Distributed Computing Theory, Morgan Ćlaypool, 125 pages, ISBN 9781681733487 (2018)