Tight space bounds for ℓ-exclusion[*]

Gadi Taubenfeld[†]

November 23, 2013

Abstract

The *ℓ-exclusion* problem is to design an algorithm which guarantees that up to ℓ processes and no more may simultaneously access identical copies of the same non-sharable resource when there are several competing processes. For $\ell = 1$, the 1-exclusion problem is the familiar mutual exclusion problem.

The simplest deadlock-free algorithm for mutual exclusion requires only one single-writer non-atomic bit per process [6, 9, 16]. This algorithm is known to be space optimal [7, 9]. For over 20 years now it has remained an intriguing open problem whether a similar type of algorithm, which uses only one single-writer bit per process, exists also for ℓ-exclusion for some $\ell \geq 2$.

We resolve this longstanding open problem. For any ℓ and n, we provide a tight space bound on the number of single-writer bits required to solve ℓ-exclusion for n processes. It follows from our results that it is not possible to solve ℓ-exclusion with one single-writer bit per process, for any $\ell \geq 2$.

In an attempt to understand the inherent difference between the space complexity of mutual exclusion and that of ℓ-exclusion for $\ell \geq 2$, we define a weaker version of ℓ-exclusion in which the liveness property is relaxed, and show that, similarly to mutual exclusion, this weaker version can be solved using one single-writer non-atomic bit per process.

Keywords: Mutual Exclusion, ℓ-exclusion, space complexity, tight bounds.

[*]A preliminary version of the results presented in this paper, appeared in *proceedings of the 25th international symposium on distributed computing* (DISC 2011), Rome, Italy, September 2011. In: LNCS 6950 Springer Verlag 2011, 110–124.

[†]The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel. tgadi@idc.ac.il

1 Introduction

1.1 Motivation

The *ℓ-exclusion* problem, which is a natural generalization of the mutual exclusion problem, is to design an algorithm which guarantees that up to ℓ processes and no more may simultaneously access identical copies of the same non-sharable resource when there are several competing processes. A solution is required to withstand the slow-down or even the crash (fail by stopping) of up to $\ell - 1$ of the processes. A process that fails by crashing simply stops executing more steps of its program, and hence, there is no way to distinguish a crashed process from a correct process that is running very slowly. For $\ell = 1$, the 1-exclusion problem is the familiar mutual exclusion problem.

A good example, which demonstrates why a solution for mutual exclusion does not also solve ℓ-exclusion (for $\ell \geq 2$), is that of a bank where people are waiting for a teller. Here the processes are the people, the resources are the tellers, and the parameter ℓ is the number of tellers. We notice that the usual bank solution, where people line up in a single queue, and the person at the head of the queue goes to any free teller, does *not* solve the ℓ-exclusion problem. If $\ell \geq 2$ tellers are free, a proper solution should enable the first ℓ people in line to move simultaneously to a teller. However, the bank solution, requires them to move past the head of the queue one at a time. Moreover, if the person at the front of the line "fails", then the people behind this person wait forever. Thus, a better solution is required which will not let a single failure tie up all the resources.

The simplest deadlock-free algorithm for mutual exclusion, called the One-bit algorithm, requires only one single-writer non-atomic shared bit per process [6, 9, 16]. The One-bit algorithm is known to be space optimal [7, 9]. For over 20 years now it has remained an intriguing open problem whether a similar type of algorithm, which uses only one single-writer bit per process, exists for ℓ-exclusion for some $\ell \geq 2$. In [19], Peterson refers to the One-bit algorithm, and writes: "Unfortunately, there seems to be no obvious generalization of their algorithm to ℓ-exclusion in general". He further points out that it is an interesting open question whether this can be done even for $n = 3$ and $\ell = 2$, where n is the number of processes. This problem is one of the oldest open problems in concurrent computing.

In this paper we resolve this longstanding open problem. For any ℓ and n, we provide a tight space bound on the number of single-writer bits required to solve the ℓ-exclusion problem for n processes. It follows from our results that it is possible to solve the problem with one single-writer bit per process, only in the case where $\ell = 1$.

1.2 The ℓ-exclusion Problem

To illustrate the ℓ-exclusion problem, consider the case of buying a ticket for a bus ride. Here a resource is a seat on the bus, the parameter ℓ is the number of available seats, The number of people who are trying to buy tickets (the processes) is assumed to be bigger than ℓ, so the ticket buyers are competing for the ℓ available seats. In the ℓ-exclusion problem, a passenger needs only to make sure that there is some free seat on the bus, but not to reserve a particular seat. A stronger version, called ℓ-assignment (or slotted ℓ-exclusion), would also require to reserve a particular seat.

More formally, it is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section* and *exit*. The ℓ-exclusion problem is to write the code for the *entry code* and the *exit code* in such a way that the following basic requirements are satisfied.

ℓ-exclusion: *No more than ℓ processes are at their critical sections at the same time.*

A *faulty* process is a process that is delayed forever. More formally, we say that a process p *fails* in an infinite run y, if there exists a finite prefix x of y such that p is not in its remainder at the end of x and the suffix of y obtained by removing x from y, does not involve p. We shall use the terms, failed process and crashed process, interchangeably.

ℓ-deadlock-freedom: *If strictly fewer than ℓ processes fail and a non-faulty process is trying to enter its critical section, then some non-faulty process eventually enters its critical section.*

We notice that the above standard definition of the ℓ-deadlock-freedom requirement is (slightly) stronger than only requiring that "if fewer than ℓ processes are in their critical sections, then it is possible for another process to enter its critical section, even though no process leaves its critical section in the meantime".

The ℓ-deadlock-freedom requirement may still allow "starvation" of individual processes. It is possible to consider stronger progress requirements which do not allow starvation. In the sequel, by an ℓ-exclusion algorithm we mean an algorithm that satisfies both ℓ-exclusion and ℓ-deadlock-freedom. We also make the standard requirements that (1) the exit code is required to be *wait-free*: once a non-faulty process starts executing its exit code, it always finishes it regardless of the activity of the other processes, and (2) a non-faulty process cannot stay in its critical section forever.

In an attempt to pinpoint the reason for the inherent difference between the space complexity of mutual exclusion and that of ℓ-exclusion for $\ell \geq 2$, we will also consider a weaker version of the ℓ-exclusion problem in which the liveness property is relaxed. Let n be the number of processes,

Weak ℓ-deadlock-freedom: *If strictly fewer than ℓ processes fail, at least one non-faulty process is trying to enter its critical section, and at least $n - \ell$ processes are in their remainders, then some non-faulty process eventually enters its critical section, provided that no process leaves its remainder in the meantime.*

The weak ℓ-deadlock-freedom property guarantees that as long as no more than ℓ processes try to enter their critical sections, all non-faulty processes should succeed regardless of how many processes crash. By a *weak* ℓ-exclusion algorithm we mean an algorithm that satisfies (1) ℓ-exclusion, (2) 1-deadlock-freedom, and (3) weak ℓ-deadlock-freedom. For $\ell = 1$, a weak 1-exclusion algorithm is a mutual exclusion algorithm.

1.3 Results

Our model of computation consists of an asynchronous collection of n processes that communicate only by reading and writing *single-writer* registers. A single-writer register can be written by one predefined process and can be read by all the processes. A register can be atomic or non-atomic. With an atomic register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action. When reading or writing a non-atomic register, a process may be reading a register while another is writing into it, and in that event, the value returned to the reader is arbitrary [15]. Our results are:

A space lower bound. For any $\ell \geq 2$ and $n > \ell$, any ℓ-exclusion algorithm for n processes must use at least $2n - 2$ bits: at least two bits per process for $n - 2$ of the processes and at least one bit per process for the remaining two processes. (Here a bit can be atomic or non-atomic.)

A matching space upper bound. For $\ell \geq 2$ and $n > \ell$, there is an ℓ-exclusion algorithm for n processes that uses $2n - 2$ non-atomic bits: two bits per process for $n - 2$ of the processes and one bit per process for the remaining two processes.

An optimal weak ℓ-exclusion algorithm. For $\ell \geq 2$ and $n > \ell$, there is a weak ℓ-exclusion algorithm for n processes that uses one non-atomic bit per process.

1.4 Related Work

To place our results in perspective, we give a brief history of the ℓ-exclusion problem. The mutual exclusion problem was first stated and solved for n processes by Dijkstra in [10]. Numerous solutions for the problem have been proposed since it was first introduced in 1965. Because of its importance and as a result of new hardware and software developments, new solutions to the problem are being designed all the time.

In [7, 9], Burns and Lynch have shown that any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers, even when *multi-writer* registers are allowed. This important lower bound can be easily generalized to show that also any ℓ-exclusion algorithm for n processes must use at least n shared registers for any $\ell \geq 1$. The One-bit mutual exclusion algorithm, which uses n non-atomic bits and hence provides a tight space upper bound, was developed independently by Burns [6] (also appeared in [9]), and by Lamport [16].

The ℓ-exclusion problem, which generalizes the mutual exclusion problem, was first defined and solved in [12, 13]. For a model which supports read-modify-write registers, a tight space bound of $\Theta(n^2)$ shared states is proved in [12, 13], for the ℓ-exclusion problem *for fixed* ℓ assuming the strong FIFO-enabling liveness property (and strong robustness). There is a large gap between the constants in the upper and lower bounds. Both depend on ℓ, but the constant in the upper bound is exponential in ℓ, while the constant in the lower bound is linear in ℓ. Without the strong liveness property and when not requiring that the exit code be wait-free, $O(n)$ states suffice for mutual exclusion using read-modify-write registers [18]. Several algorithms for ℓ-exclusion, which are based on strong primitives such as fetch-and-increment and compare-and-swap, are considered in [3]. The "bank example" in Section 1.1 is from [13].

In [19], Peterson has proposed several ℓ-exclusion algorithms for solving the problem using atomic read/write registers satisfying various progress properties ranging from ℓ-deadlock-freedom to FIFO-enabling. The simplest algorithm in [19] requires a single 3-valued single-writer atomic register per process. A FIFO-enabling ℓ-exclusion algorithm using atomic registers is presented also in [1], which makes use of concurrent time-stamps for solving the problem with bounded size memory. Long-lived and adaptive algorithms for collecting information using atomic registers are presented in [2]. The authors employ these algorithms to transform the ℓ-exclusion algorithm of [1], into its corresponding adaptive long-lived version. An ℓ-exclusion algorithm using $O(n^2)$ non-atomic single-writer bits which does not permits individual starvation of processes, is presented in [11].

In [8], an interesting tight bound regarding the ℓ-assignment problem is proved. Namely, that $\ell \geq 2k + 1$ is a necessary and sufficient condition for solving ℓ-assignment problem using atomic registers where k is the maximal number of possible faults. In [4], a simpler version of the ℓ-assignment problem, called *distinct CS*, is considered. Finally, in [5], the ℓ-exclusion problem (called in [5] the "identical-slot critical section" problem) is considered in a completely asynchronous distributed network. The message complexity of the solution is unbounded. Many known mutual exclusion, ℓ-exclusion and ℓ-assignment algorithms are discussed in detail in [20].

2 A Space Lower Bound

In this section we assume a model where the only shared objects are atomic registers. It is obvious that the space lower bound applies also for non-atomic registers. In the following, by a *register* (*bit*), we mean an atomic single-writer register (bit).

Theorem 2.1 *For any $\ell \geq 2$ and $n > \ell$, any ℓ-exclusion algorithm for n processes must use at least $2n - 2$ bits: at least two bits per process for $n - 2$ of the processes and at least one bit per process for the remaining two processes.*

In the next section we will provide a matching upper bound. Interestingly, Theorem 2.1 follows from the following special case when $\ell = 2$ and $n = 3$.

Theorem 2.2 *Any 2-exclusion algorithm for 3 processes must use at least 4 bits: at least two bits for one of the processes and at least one bit for each one of the remaining two processes.*

Proof of Theorem 2.1: We observe that any ℓ-exclusion algorithm, say A, where $\ell \geq 2$ for processes $p_1, ..., p_n$ (where $n > \ell$), can be transformed into a 2-exclusion algorithm, say A', for processes p_1, p_2, p_3, where the space for each one of the three processes in A' is the same as the space for the corresponding process in A. Let the $\ell - 2$ processes $p_4, ..., p_{\ell+1}$ execute A until they enter their critical sections and then let them crash, and let st be the global state immediately after these $\ell - 2$ processes crash (by the ℓ-deadlock-freedom requirement, such an execution exists). A' is now constructed from A by replacing each one of the single-writer registers of processes $p_4, ..., p_n$, with a constant whose value equals the value of the corresponding register in state st. The programs of processes p_1, p_2, p_3 in A' are the same as in A, except the fact that in A' whenever a process needs to read the value of a register of one of the processes $p_4, ..., p_n$, it does so by accessing the corresponding constant.

It follows from Theorem 2.2 and the above transformation that, for *any three* processes from the set of n processes which participate in A, one of three processes must "own" at least two bits and each one of remaining two processes must "own" at least one bit. Thus, the n processes together must use at least $2n - 2$ bits: two bits per process for $n - 2$ of the processes and one bit per process for the remaining two processes. ▮

For the rest of the section, we focus on proving Theorem 2.2.

2.1 A Formal Model

Our model of computation, for proving the lower bound, consists of an asynchronous collection of n deterministic processes that communicate via single-writer atomic registers. An *event* corresponds to an atomic step performed by a process. The events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but do not return a value. An event specifies which process performs it, a read event specifies the value that is read from a register, and a write event specifies the value that is written into a register. A (global) *state* of an algorithm is completely described by the values of the registers and the values of the location counters of all the processes.

A *run* is a sequence of alternating states and events (also referred to as steps). For the purpose of the lower bound proof, it is more convenient to define a run as a sequence of events omitting all the states except the initial state. Since the states in a run are uniquely determined by the events and the initial state, no information is lost by omitting the states. Each event in a run is associated with a process that is *involved* in the event. We use the notation e_p to denote an instance of an arbitrary event at a process p.

We will use x, y and z to denote runs. The notation $x \leq y$ means that x is a prefix of y, and $x < y$ means that x is a proper prefix of y. When $x \leq y$, 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. We will often use statements like "at the end of run x process p is in its remainder", and implicitly assumed that there is a function which for any finite run and process, lets us know whether a process is in its remainder, entry, critical section, or exit code, at the end of that run. Also, saying that an extension y of x involves only processes from the set P means that all events in $(y - x)$ are only by processes in P. It is assumed that the processes are deterministic. That is, if $x; e_p$ and $x; e'_p$ are runs then $e_p = e'_p$.

Next we define the *looks like* relation which captures when two runs are indistinguishable to a given process.

Definition 2.1 *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 at the end of x are the same as at the end of y.*

The looks like relation is also called the indistinguishability relation in the literature. The looks like relation is an equivalence relation for a given process p. The next step by a process always depends on the previous step taken by the process and the current values of the registers. It should be clear that if two runs look alike to a given process then the next step by this process at the end both runs is the same.

Lemma 2.3 *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 that 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 involves in e. Then, from the construction, the runs z' and y' look alike to p, which implies that the next step by p at the end of both runs is the same. Thus, since $z = z'; e$ is a run, also $y'; e = y; (z - x)$ is a run. ▮

Lemma 2.4 *In a model where only single-writer registers are used, any ℓ-exclusion algorithm, where $\ell \geq 1$, must use at least one single-writer register per process.*

Proof: Let x be a run in which all the processes are in their remainders at the end of x, and let z be an extension of x, by events which involve only process p, in which p is in its critical section (such a run z exists by the ℓ-deadlock-freedom property). We show that there must be a write event which involves p in $(z - x)$. This implies that that each process p must have at least one single-writer register.

Assume to the contrary that $(z - x)$ involves only steps by p, and there is no write event which involves p in $(z - x)$. Since none of the events in $(z - x)$ involve the other processes, z looks like x to all processes other than p. By the ℓ-deadlock-freedom property, there is an extension of x which does not involve p in which ℓ processes (not including p) enter their critical sections. Since z looks like x to all the processes other than p, by Lemma 2.3, a similar extension exists starting from z. That is, ℓ additional processes can enter their critical sections in an extension of z, while p is still in its critical section. This violates the ℓ-exclusion property. ∎

2.2 A high-level description of the proof of Theorem 2.2

Before getting into specific details, we give below a very high-level intuitive description of the lower bound proof, so that the readers can have the big picture already in their mind while following the proof.

To prove Theorem 2.2, we assume to the contrary that there exists a 2-exclusion algorithm, called $2EX$, for three processes which uses only three shared bits. By Lemma 2.4, $2EX$ uses one single-writer bit per process. We show that this assumption leads to a contradiction. This is done by constructing an infinite run in which at least two (out of the three) processes participate infinitely often and where no process ever enters its critical section. The existence of such an infinite run violates the 2-deadlock-freedom requirement.

The construction of the above infinite run is done as follows. First we introduce (in Subsection 2.4) the key concept of a *locked* process. Intuitively, process p is *locked* at the end a given run, if p *must* wait for at least one other process to take a step before it may enter its critical section. Next we prove the following two lemmas for the algorithm $2EX$:

1. There exists a run x_0, such that all three processes are locked at the end x_0, and no process is in its critical section at the end of any prefix of x_0 (Lemma 2.7).

2. For any run x, such that all three processes are locked at the end x, there exists an extension z of x such that (1) all the processes are locked at the end of z, (2) exactly two of the processes are involved in the suffix of z obtained by removing x from z, and (3) no process is in its critical section at the end of any prefix of z (Lemma 2.14).

Starting from such run x_0, we repeatedly apply the result mentioned in item (2) above, to construct the desired infinite run, in which at least two processes take infinitely many steps, but no process ever enters its critical section.

Thus, for any 2-exclusion algorithm for 3 processes which uses one single-writer bit per process, there exists an infinite run in which at least two processes take infinitely many steps, but no process ever enters its critical section. The existence of such a run implies that the algorithm does not satisfy 2-deadlock-freedom, which leads to a contradiction.

2.3 Changing the value of a bit

All the definitions and lemmas below refer to the arbitrary 2-exclusion algorithm for *three* processes, called $2EX$, which uses one single-writer bit per process.

For process p and run x, we use the notation $value(x, p)$ to denote the value at the end of run x, of the single-writer bit that only p is allowed to write into. I.e., the value of p's single-writer bit at the global state immediately after the last event of x has occurred.

Lemma 2.5 *Let P be a set of processes where $|P| \leq 2$, let z be a run in which the processes in P are in their critical sections at the end of z, and let x be the longest prefix of z such that the processes in P are in their remainders at the end of x. If $(z - x)$ involves only steps by processes in P, then $value(x, p) \neq value(z, p)$ for every $p \in P$.*

Proof: Assume to the contrary that $value(x, p) = value(z, p)$ for some $p \in P$.

Case 1: $|P| = 1 = \{p\}$. Since none of the events in $(z - x)$ involves the other two processes, x looks like z to all processes other than p. By the 2-deadlock-freedom property, there is an extension of x which does not involve p in which the other two processes enter their critical sections. Since x looks like z to all processes other than p, by Lemma 2.3, a similar extension exists starting from z. That is, the other two processes can enter their critical sections in an extension of z, while p is still in its critical section. This violates the 2-exclusion property.

Case 2: $|P| = 2 = \{p, q\}$, $value(x, p) = value(z, p)$ and $value(x, q) = value(z, q)$. Since none of the events in $(z - x)$ involves the third process, call it process r, x looks like z to r. By the 2-deadlock-freedom property, there is an extension of x which involves only r in which r enters its critical section. Since x looks like z to r, by Lemma 2.3, a similar extension exists starting from z. That is, r can enter its critical section in an extension of z, while p and q are still in their critical sections. This violates the 2-exclusion property.

Case 3: $|P| = 2 = \{p, q\}$, $value(x, p) = value(z, p)$ and $value(x, q) \neq value(z, q)$. By the 2-deadlock-free property, there is an extension y of x in which q is in its critical section at the end of y and $(y - x)$ involves only q. By case 1 above, $value(x, q) \neq value(y, q)$. Since none of the events in $(y - x)$ and in $(z - x)$ involves r, y looks like z to r. (Notice that, here we use also the fact that the shared registers are single-writer bits.) By the 2-deadlock-freedom property, there is an extension of y which involves only r in which r enters its critical section. Since y looks like z to r, by Lemma 2.3, a similar extension exists starting from z. That is, r can enter its critical section in an extension of z, while p and q are still in their critical sections. This violates the 2-exclusion property. ∎

Lemma 2.6 *Let z be a run in which p is in its remainder at the end of z, and assume that there is a prefix of z in which process p is in its critical section (at the end of that prefix). Let x be the longest prefix of z such that p is in its critical section at the end of x. If $(z - x)$ involves only steps by p, then $value(x, p) \neq value(z, p)$.*

Proof: Assume to the contrary that $(z - x)$ involves only steps by p, and $value(x, p) = value(z, p)$. Since none of the events in $(z - x)$ involves the other processes, z looks like x to all processes other than p. By the 2-deadlock-freedom property, there is an extension of z which does not involve p in which the other two processes enter their critical sections. Since z looks like x to all the processes other than p, by Lemma 2.3, a similar extension exists starting from x. That is, the other two processes can enter their critical sections in an extension of x, while p is still in its critical section. This violates the 2-exclusion property. ∎

2.4 Locking

Next we introduce the key concept of a *locked* process. Intuitively, process p is *locked* at the end of a given run, if p *must* wait for at least one other process to take a step before it may enter its critical section.

Definition 2.2 *For process p and run x, p is* ***locked*** *at the end of x, if (1) p is in its entry code at the end of x, and (2) for every extension y of x such that $(y - x)$ involves only steps by p, p is in its entry code at the end of y.*

Lemma 2.7 *There exists a run x_0, such that all three processes are locked at the end of x_0, and no process is in its critical section at the end of any prefix of x_0.*

Proof: The run x_0 is constructed as follows. We start from (any) one of the possible initial states, and denote the initial values of the three bits of the processes p_1, p_2, p_3 in the initial state by $init(p_1), init(p_2)$ and $init(p_3)$, respectively. We first let p_1 run alone until it is about to write and change the value of its single-writer bit from $init(p_1)$ to $1 - init(p_1)$ for the *last* time before it may enter its critical section if it continues to run alone (by Lemma 2.5 such a write must eventually happen). We suspend p_1 just before it writes and repeat this procedure with p_2 and then with p_3. Then we let the processes p_1, p_2, p_3 write the values $1 - init(p_1), 1 - init(p_2), 1 - init(p_3)$, respectively, into their bits. The resulting run, where all three bits are set to values which are different from their initial values, is x_0. Each process $p \in \{p_1, p_2, p_3\}$, cannot distinguish x_0 from a run in which before p has executed this last write, the other two processes run alone, flipped the values of their bits (by Lemma 2.5), and have entered their critical sections. Thus, it follows from the 2-exclusion property and Lemma 2.3, that there cannot be an extension of x_0 by steps of p only in which p is in its critical section, which implies that each one of the three processes is locked at the end of x_0. ▮

Lemma 2.8 *Assume that x looks like z to p. Process p is locked at the end of x if and only if p is locked at the end of z.*

Proof: Assume to the contrary that p is locked at the end of x and p is *not* locked at the end of z. By definition, there is an extension $\hat{z}$ of z such that $(\hat{z} - z)$ involves only p and p is in its critical section at the end of $\hat{z}$. Since x looks like z to p, by Lemma 2.3, a similar extension exists starting from x. That is, p is in its critical section at the end of $x; (\hat{z} - z)$. This contradicts the assumption the p is locked at the end of x. Since the *looks like* relation is symmetric, the result follows. ▮

Lemma 2.9 *Assume that p is in its remainder at the end of x. For every $q \neq p$, q is not locked at the end of x.*

Proof: This follows immediately from the 2-deadlock-freedom requirement. ▮

2.5 Locking and Values

The following lemmas relate between locked processes and the values of their bits.

Lemma 2.10 *Assume that the three processes are locked at the end of x, and let z be an extension of x such that some process p is not involved in $(z - x)$ and some other process q is in its remainder at the end of z. Then, $value(x, q) \neq value(z, q)$.*

Proof: Assume to the contrary that $value(x, q) = value(z, q)$. Let r be the third process. By the 2-deadlock-freedom requirement, there is an extension $\hat{z}$ of z such that (1) $(\hat{z} - z)$ involves only r, and (2) r is in its critical section at the end of $\hat{z}$. There are two cases.

Case 1: $value(x, r) = value(\hat{z}, r)$. In this case, x looks like $\hat{z}$ to p. By Lemma 2.8, p is locked at the end of $\hat{z}$. This violates Lemma 2.9 (i.e., this violates the 2-exclusion property).

Case 2: $value(x, r) \neq value(\hat{z}, r)$. By Lemma 2.6 and the assumption that the exit code is wait-free, there is an extension z' of $\hat{z}$, such that (1) $(z' - \hat{z})$ involves only r, (2) r is in its remainder at the end of z', and (3) $value(\hat{z}, r) \neq value(z', r)$. Thus, $value(x, r) = value(z', r)$. This implies that x looks like z' to p. By Lemma 2.8, p is locked at the end of z'. This violates Lemma 2.9. Thus, we have reached a contradiction. The result follows. ∎

Lemma 2.11 *Assume that the three processes are locked at the end of x, and let z be an extension of x such that some process p is not involved in $(z - x)$ and some other process q is in its critical section at the end of z. Then, $value(x, q) = value(z, q)$.*

Proof: By Lemma 2.6 and the assumption that the exit code is wait-free, there is an extension z' of z, such that (1) $(z' - z)$ involves only q, (2) q is in its remainder at the end of z', and (3) $value(z, q) \neq value(z', q)$. By Lemma 2.10, $value(x, q) \neq value(z', q)$ Thus, $value(x, q) = value(z, q)$. ∎

Lemma 2.12 *Assume that the three processes are locked at the end of x. For every two processes p and q there is an extension z of x such that:* (1) $(z - x)$ *involves only p and q,* (2) *p is in its critical section at the end of z,* (3) *q is in its remainder at the end of z,* (4) $value(x, p) = value(z, p)$*, and* (5) $value(x, q) \neq value(z, q)$.

Proof: We construct z as follows. Starting from x we first let p and q run until one of them enters its critical section. This is possible by the 2-deadlock-freedom requirement. Then, we let the process that has entered its critical section, say process q, continue alone until it reaches its remainder (recall that the exit code is assumed to be wait-free). At that point, we let the other process, say process p, run alone until it enters its critical section (again, this is possible by the 2-deadlock-freedom requirement). The resulting run, at the end of which p is in its critical section, and q is in its remainder is z. We notice that only p and q are involved in $(z - x)$. By Lemma 2.11, $value(x, p) = value(z, p)$, and by Lemma 2.10, $value(x, q) \neq value(z, q)$. ∎

2.6 Flexibility

Intuitively, process p is *flexible* at the end of a given run, if p *can* change the value of its bit without a need to wait for some other process to take a step.

Definition 2.3 *For process p and run x, p is **flexible** at the end of x, if there exists an extension z of x such that: (1) $(z - x)$ involves only steps by p, (2) for every $x \leq y \leq z$, p is in its entry code at the end of y, and (3) $value(x, p) \neq value(z, p)$.*

Lemma 2.13 *Let x be a run such that the three processes are locked at the end of x. Then, at least two processes are flexible at the end of x.*

Proof: We assume that p_1, p_2 and p_3 are locked at the end of x. By the 2-deadlock-freedom property, if we extend x by letting processes p_1 and p_2 taking steps alternately, eventually one of the two must enter its critical section. Let $\hat{z}$ be the shortest extension of x such that at the end of $\hat{z}$ both processes are still in their entry code, but the next step of one of them is already in a critical section.

Case 1: for some y where $x \leq y \leq \hat{z}$, either $value(x, p_1) \neq value(y, p_1)$ or $value(x, p_2) \neq value(y, p_2)$. Let y be the *shortest* run where $x < y \leq \hat{z}$, and $value(x, p_1) \neq value(y, p_1)$ or $value(x, p_2) \neq value(y, p_2)$. This means that only one process, say p_1, changed its bit in y, and all the steps of p_2 in $(y - x)$ are invisible to p_1 and p_3. Thus, it is possible to remove from y all the events in which p_2 is involved in $(y - x)$ and get a new run z. Clearly z looks like y to p_1 and hence $value(y, p_1) = value(z, p_1)$, which implies that $value(x, p_1) \neq value(z, p_1)$. Thus, z is a *witness* for the fact that p_1 is flexible at the end of x.

Case 2: for all y where $x \leq y \leq \hat{z}$, $value(x, p_1) = value(y, p_1)$ and $value(x, p_2) = value(y, p_2)$. Thus, all the steps of the p_1 in $(\hat{z} - x)$ are invisible to p_2 and p_3, and all the steps of the p_2 in $(\hat{z} - x)$ are invisible to p_1 and p_3. Thus, it is possible to remove from $\hat{z}$ all the events in which p_2 is involved in $(\hat{z} - x)$ and get a new run z_1, and similarly, it is possible to remove from $\hat{z}$ all the events in which p_1 is involved in $(\hat{z} - x)$ and get a new run z_2. Clearly, z_1 looks like $\hat{z}$ to p_1 and z_2 looks like $\hat{z}$ to p_2. By definition, either the next step of p_1 from z_1 is already a step in its critical section, or the next step of p_2 from z_2 is already a step in its critical section. This contradicts the assumption that all the processes are locked at the end of x.

We conclude that either p_1 or p_2 is flexible at the end of x. Assume without loss of generality that p_1 is flexible at the end of x. We repeat the above argument to show that either p_2 or p_3 is flexible at the end of x. ∎

2.7 Main Lemma and Proof of Theorem 2.2

We are now ready to prove the main lemma and complete the proof of Theorem 2.2.

Lemma 2.14 (main lemma) *Let x be a run such that all three processes are locked at the end of x. Then, there exists an extension z of x such that (1) all the processes are locked at the end of z, (2) exactly two of the processes are involved in $(z - x)$, and (3) for every $x \leq y \leq z$, all the processes are in their entry codes at the end of y.*

Proof: We will construct the run z and show that z satisfies the required properties. We start from the run x in which the three processes, p_1, p_2, p_3 are locked at the end of x. By Lemma 2.13 at least two processes are flexible at the end of x. Without loss of generality, we assume that p_2 and p_3 are flexible at the end of x. We construct an extension of x, called x_2, which involves only process p_2 as follows. Starting from x, we let p_2 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_2)$ to $1 - value(x, p_2)$. We suspend p_2 just *before* it writes, and call this run x_2. For later reference, we denote by e_{p_2} the write event that p_2 is about to take once activated again, and denote by x'_2 the run $x_2; e_{p_2}$. See Figure 1 for illustration of all the runs which are constructed for the proof of Lemma 2.14.

Similarly, we construct an extension of x, called x_3, which involves only process p_3 as follows. Starting from x, we let p_3 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_3)$ to $1 - value(x, p_3)$. We suspend p_3 just *before* it writes, and call this run x_3. For later reference, we denote by e_{p_3} the write event that p_3 is about to take once activated again, and denote by x'_3 the run $x_3; e_{p_3}$.

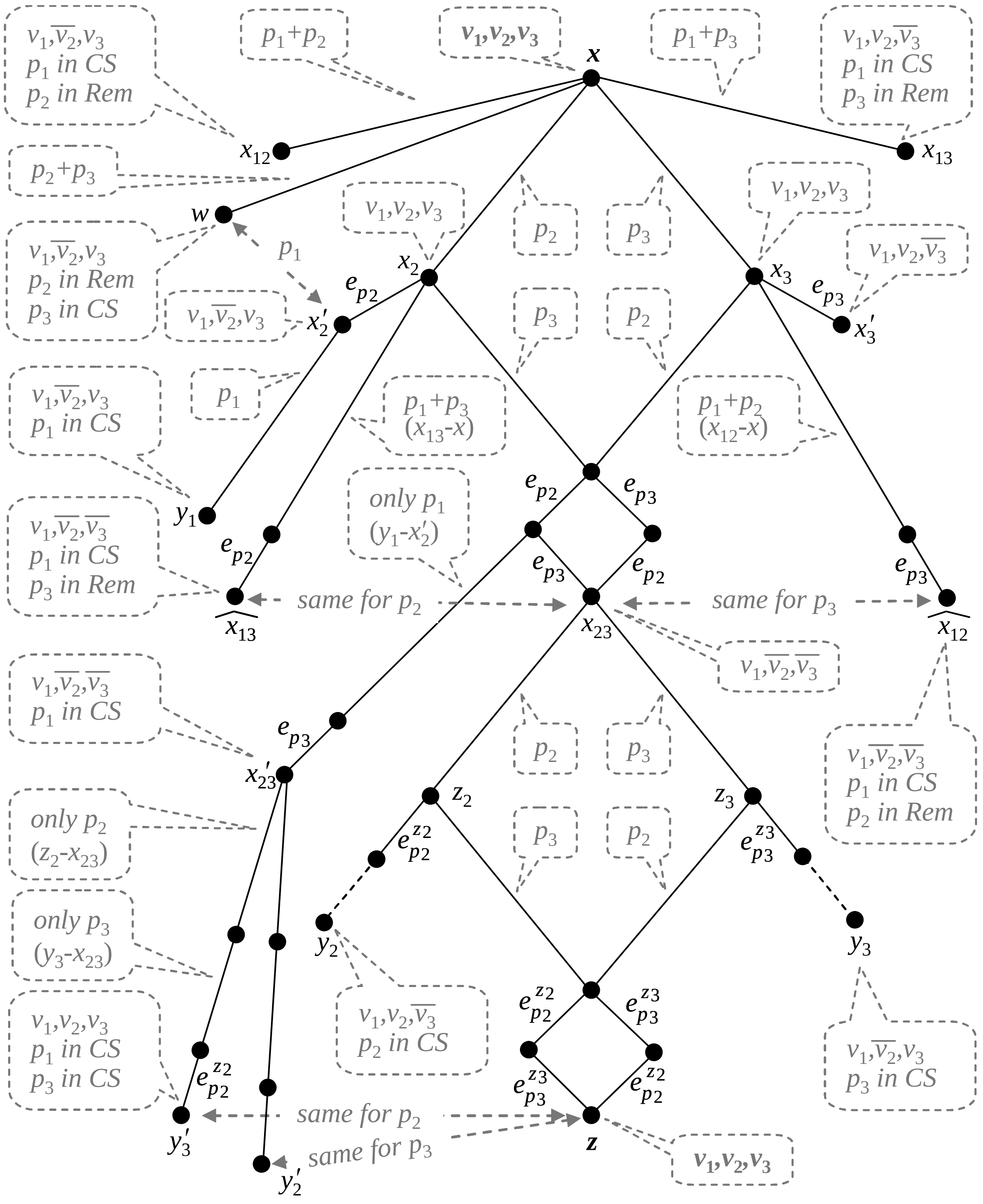
$v_1, \overline{v_2}, v_3$ p_1 in CS p_2 in Rem
p_1+p_2
v_1, v_2, v_3
x
p_1+p_3
$v_1, v_2, \overline{v_3}$ p_1 in CS p_3 in Rem
x_{12}
x_{13}
p_2+p_3
w
$v_1, \overline{v_2}, v_3$ p_2 in Rem p_3 in CS
p_1
v_1, v_2, v_3
x_2
e_{p_2}
x_2'
$v_1, \overline{v_2}, v_3$
p_2
p_3
p_3
p_2
v_1, v_2, v_3
x_3
e_{p_3}
x_3'
$v_1, v_2, \overline{v_3}$
p_1
$v_1, \overline{v_2}, v_3$ p_1 in CS
y_1
p_1+p_3 $(x_{13}-x)$
p_1+p_2 $(x_{12}-x)$
only p_1 (y_1-x_2')
e_{p_2}
e_{p_3}
$v_1, \overline{v_2}, \overline{v_3}$ p_1 in CS p_3 in Rem
$\widehat{x_{13}}$
same for p_2
x_{23}
same for p_3
$\widehat{x_{12}}$
$v_1, \overline{v_2}, \overline{v_3}$
$v_1, \overline{v_2}, \overline{v_3}$ p_1 in CS
x_{23}'
$v_1, \overline{v_2}, \overline{v_3}$ p_1 in CS p_2 in Rem
z_2
z_3
$e_{p_2}^{z_2}$
$e_{p_3}^{z_3}$
only p_2 (z_2-x_{23})
only p_3 (y_3-x_{23})
y_2
y_3
$v_1, v_2, \overline{v_3}$ p_2 in CS
$v_1, \overline{v_2}, v_3$ p_3 in CS
v_1, v_2, v_3 p_1 in CS p_3 in CS
y_3'
y_2'
same for p_2
same for p_3
z
v_1, v_2, v_3

Figure 1: Illustration of all the runs in the proof of Lemma 2.14. For $i \in \{1, 2, 3, \}, v_i = value(x, p_i)$ and $\overline{v_i} = 1 - v_i$. *Rem* and *CS* are abbreviations for *Remainder* and *Critical Section*, respectively.

Next we construct an extension of x, called x_{23}, which involves only p_2 and p_3. We let p_2 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_2)$ to $1 - value(x, p_2)$. We suspend p_2 just before it writes and then let p_3 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_3)$ to $1 - value(x, p_3)$. We suspend p_3 just before it writes. Then we let the processes p_2 and p_3 write the values $1 - value(x, p_2)$ and $1 - value(x, p_3)$ respectively, into their bits in an arbitrary order. The resulting run, where the bits of p_2 and p_3 are set to values which are different from their values at the end of x, is run x_{23}. Thus, the run x_{23} is the run $x_2; (x_3 - x); e_{p_2}; e_{p_3}$ (the order in which e_{p_2} and e_{p_3} are executed is immaterial). We observe that $x_2; (x_3 - x)$ looks like $x_3; (x_2 - x)$ to all the processes.

We consider also the following two extensions of x. By Lemma 2.12, there exists an extension of x which we will call run x_{12}, such that $(x_{12} - x)$ involves only p_1 and p_2, p_1 is in its critical section at the end of x_{12}, p_2 is in its remainder at the end of x_{12}, $value(x, p_1) = value(x_{12}, p_1)$, and $value(x, p_2) \neq value(x_{12}, p_2)$. By Lemma 2.12, there exists another extension of x which we will call run x_{13}, such that $(x_{13} - x)$ involves only p_1 and p_3, p_1 is in its critical section at the end of x_{13}, p_3 is in its remainder at the end of x_{13}, $value(x, p_1) = value(x_{13}, p_1)$, and $value(x, p_3) \neq value(x_{13}, p_3)$.

We notice that $(x_{12} - x)$ must include a write event by p_2, and the last write event by p_2 in $(x_{12} - x)$ sets the value of p_2 single-writer bit to $1 - value(x, p_2)$. Similarly, $(x_{13} - x)$ must include a write event by p_3, and the last write event by p_3 in $(x_{13} - x)$ sets the value of p_3 single-writer bit to $1 - value(x, p_3)$.

Let the run $\widehat{x_{12}}$ be the run $x; (x_3 - x); (x_{12} - x); e_{p_3}$, and let the run $\widehat{x_{13}}$ be the run $x; (x_2 - x); (x_{13} - x); e_{p_2}$. Clearly, x_{23} looks like $\widehat{x_{12}}$ to p_3, and x_{23} looks like $\widehat{x_{13}}$ to p_2. Informally, this implies that at the end of x_{23}, process p_2 "suspects" that p_1 is in its critical section and p_3 is in its remainder; and process p_3 "suspects" that p_1 is in its critical section and p_2 is in its remainder. Since p_2 is in its remainder at the end of $\widehat{x_{12}}$, by Lemma 2.9, p_3 is not locked at the end of $\widehat{x_{12}}$, and hence by Lemma 2.8, p_3 is also not locked at the end of x_{23}. Similarly, since p_3 is in its remainder at the end of $\widehat{x_{13}}$, by Lemma 2.9, p_2 is not locked at the end of $\widehat{x_{13}}$, and hence by Lemma 2.8, p_2 is also not locked at the end of x_{23}.

Since both p_2 and p_3 are not locked at the end of x_{23}, there is an extension y_2 of x_{23} by steps of p_2 only in which p_2 is in its critical section at the end of y_2, and there is (another) extension y_3 of x_{23} by steps of p_3 only in which p_3 is in its critical section at the end of y_3. Since both $(y_2 - x)$ and $(y_3 - x)$ do not involve p_1, by Lemma 2.11, $value(x, p_2) = value(y_2, p_2)$ and $value(x, p_3) = value(y_3, p_3)$. Thus, $value(x_{23}, p_2) \neq value(y_2, p_2)$ and $value(x_{23}, p_3) \neq value(y_3, p_3)$. This means that in $(y_2 - x_{23})$ there is a write event by p_2, denoted $e_{p_2}^{z_2}$, which changes the value of the bit of p_2 to $value(x, p_2)$, and in $(y_3 - x_{23})$ there is a write event by p_3, denoted $e_{p_3}^{z_3}$, which changes the value of the bit of p_3 to $value(x, p_3)$. Let run z_2 be the shortest extension of x_{23} such that $z_2; e_{p_2}^{z_2}$ is a prefix of y_2, and let run z_3 be the shortest extension of x_{23} such that $z_3; e_{p_3}^{z_3}$ is a prefix of y_2.

Next we construct the extension z of x_{23} which involves only p_2 and p_3. We first let p_2 run alone until it is about to change the value of its single-writer bit to $value(x, p_2)$. We suspend p_2 just before it writes and then let p_3 run alone until it is about to change the value of its single-writer bit to $value(x, p_3)$. We suspend p_3 just before it writes. Then we let the processes p_2 and p_3 write the values $value(x, p_2)$ and $value(x, p_3)$ respectively, into their bits in an arbitrary order. The resulting run, where the bits of p_2 and p_3 are set to values which are the same as their values at the end of x, is the run z. That is, z is exactly the run $z_2; (z_3 - x_{23}); e_{p_2}^{z_2}; e_{p_3}^{z_3}$ (the order in which the last two write events are executed is immaterial). Below we prove that the three processes are locked at the end of z.

Since $(z - x)$ does not involve p_1, x looks like z to p_1, and thus by Lemma 2.8, p_1 is locked at the end of z. To prove that the also p_2 and p_3 are locked at the end of x, we will show that each one of them cannot distinguish between z and another run at the end of which the other two processes are in their critical sections. We now construct these two runs.

By Lemma 2.12, there exists an extension of x which we will call run w, such that $(w - x)$ involves only p_2 and p_3, p_3 is in its critical section at the end of w, p_2 is in its remainder at the end of w, $value(x, p_3) = value(w, p_3)$, and $value(x, p_2) \neq value(w, p_2)$. Clearly w looks like x'_2 to p_1. Since p_2 is in its remainder at the end of w, by Lemma 2.9, p_1 is not locked at the end of w, and hence by Lemma 2.8, p_1 is also not locked at the end of x'_2. Thus, there is an extension y_1 of x'_2 by steps of p_1 only, in which p_1 is in its critical section at the end of y_1. Since $x'_2 - x$ does not involve steps by p_1, $value(x, p_1) = value(x'_2, p_1)$. Since $(y_1 - x)$ does not involve steps by p_3, by Lemma 2.11, $value(x, p_1) = value(y_1, p_1)$. Thus also $value(x'_2, p_1) = value(y_1, p_1)$.

Recall that $x_{23} = x_2; (x_3 - x); e_{p_2}; e_{p_3}$. We observe that (1) $(x_3 - x)$ involves only p_3 and does not change the value of the bit of p_3, and (2) $(y_1 - x'_2)$ involves only p_1 and does not change the value of the bit of p_1. Thus, by Lemma 2.3 (applied several times), the sequence of events, $x'_{23} = x_2; (x_3 - x); e_{p_2}; (y_1 - x'_2); e_{p_3}$ is a legal run in which p_1 is in its critical section at the end of this legal run.

Since x'_{23} looks like x_{23} to p_2 and to p_3, by Lemma 2.3, $x'_{23}; (y_3 - x_{23})$ is a run in which both p_1 and p_3 are in their critical sections at the end of the run, and $x'_{23}; (z_2 - x_{23})$ is a legal run. Since $(z_2 - x_{23})$ involves only p_2 and the value of the bit of p_3 does not change, by Lemma 2.3, the sequence $y'_3 = x'_{23}; (z_2 - x_{23}); (y_3 - x_{23}); e^{z2}_{p2}$ is a legal run in which both p_1 and p_3 are in their critical sections at the end of this legal run. Process p_2, cannot distinguish z from a run in which before p_2 executed its last write in z, the other two processes have entered their critical sections. That is, z looks like y'_3 to p_2. Thus, it follows from the 2-exclusion property and Lemma 2.3, that there cannot be an extension of z by steps of p_2 only in which p_2 is in its critical section at the end of this extension. This implies that p_2 is locked at the end of z.

Similarly, since x'_{23} looks like x_{23} to p_2 and to p_3, by Lemma 2.3, $x'_{23}; (y_2 - x_{23})$ is a run in which both p_1 and p_2 are in their critical sections at the end of the run, and $x'_{23}; (z_3 - x_{23})$ is a legal run. Since $(z_3 - x_{23})$ involves only p_3 and the value of the bit of p_3 does not change, by Lemma 2.3, the sequence $y'_2 = x'_{23}; (z_3 - x_{23}); (y_2 - x_{23}); e^{z3}_{p3}$ is a legal run in which both p_1 and p_2 are in their critical sections at the end of the run. Process p_3, cannot distinguish z from a run in which before p_3 executed its last write in z, the other two processes have entered their critical sections. That is, z looks like y'_2 to p_3. Thus, it follows from the 2-exclusion property and Lemma 2.3, that there cannot be an extension of z by steps of p_3 only in which p_3 is in its critical section at the end of the extension. This implies that p_3 is locked at the end of z.

To conclude, we have shown that all three processes are locked at the end of z. Furthermore, it follows from the construction that two of the processes, p_2 and p_3, are involved in $(z - x)$, and that for every $x \leq y \leq z$, all the processes are in their entry codes at the end of y. ∎

Proof of Theorem 2.2: We have assumed to the contrary that $2EX$ is a 2-exclusion algorithm for 3 processes which uses one single-writer bit per process. We show that this leads to a contradiction. By Lemma 2.7, there exists a run x_0, such that all three processes are locked at the end of x_0, and no process is in its critical section at the end of any prefix of x_0. Starting from such a run x_0 (which exists by Lemma 2.7), we repeatedly apply the result of Lemma 2.14 to construct the desired infinite run, in which at least two processes take infinitely many steps, but no process ever enters its critical section. That is, we begin with x_0 and pursue the following *locking-preserving scheduling* discipline:

```
1 $x := x_0$;                                        /* initialization */
2 repeat
3   let $z$ be an extension of $x$. The existence of $z$ is proved in Lemma 2.14, where all the
    processes are locked at the end of $z$, two processes are involved in $(z - x)$, and
    for every $x \leq y \leq z$, all the processes are in their entry codes at the end of $y$.
4   $x := z$                                /* "locking extension" of $x$ */
5 forever
```

The above scheduling discipline, implies that there is an infinite run of $2EX$ in which at least two processes take infinitely many steps, but no process ever enters its critical section. The existence of such a run implies that $2EX$ does not satisfy 2-deadlock-freedom. A contradiction. ▮

3 A Space Upper Bound: The Two-bits Algorithm

In this section we provide a tight space upper bound for ℓ-exclusion. To make the upper bound as strong as possible, we will assume that the registers are non-atomic.

Theorem 3.1 *For $\ell \geq 2$ and $n > \ell$, there is an ℓ-exclusion algorithm for n processes that uses $2n - 2$ non-atomic bits: two bits per process for $n - 2$ of the processes and one bit per process for the remaining two processes.*

We present below a space optimal algorithm which is inspired by Peterson's ℓ-exclusion algorithm which requires one 3-valued single-writer *atomic* register per process [19].

3.1 The Two-bits Algorithm

Our algorithm is for n processes each with unique identifier taken from the set $\{1, ..., n\}$. For each process $i \in \{2, ..., n-1\}$, the algorithm requires two single-writer non-atomic bits, called $Flag_1[i]$ and $Flag_2[i]$. For process 1 the algorithm requires one single-writer non-atomic bit, called $Flag_1[1]$, and for process n the algorithm requires one single-writer non-atomic bit, called $Flag_2[n]$. In addition two local variables, called *counter* and j, are used for each process. ℓ is used as a constant.

THE TWO-BITS ℓ-EXCLUSION ALGORITHM: process $i \in \{1, ..., n\}$ program.

Shared: $Flag_1[1..n-1]$, $Flag_2[2..n]$: arrays of non-atomic bits, initially all entries are 0.
Local: $counter, j$: integer ranges over $\{0, ..., n\}$.
Constant: $Flag_1[n] = 0$, $Flag_2[1] = 0$.   /* used for simplifying the presentation */

```
1   if $i \neq n$ then $Flag_1[i] := 1$ fi;                          /* save one bit */
2   repeat
3     repeat
4       $counter := 0$;
5       for $j := 1$ to $n$ do
6         if ($j < i$ and $Flag_1[j] = 1$) or ($Flag_2[j] = 1$)
7         then $counter := counter + 1$ fi od
8     until $counter < \ell$;
```

```
9       if $i \neq 1$ then $Flag_2[i] := 1$ fi;                          /* save one bit */
10      $counter := 0$;
11      for $j := 1$ to $n$ do
12         if ($j < i$ and $Flag_1[j] = 1$) or ($j \neq i$ and $Flag_2[j] = 1$)
13         then $counter := counter + 1$ fi od
14      if $counter \geq \ell$ then if $i \neq 1$ then $Flag_2[i] := 0$ fi fi
15   until $counter < \ell$;
16   critical section;
17   if $i \neq 1$ then $Flag_2[i] := 0$ fi; if $i \neq n$ then $Flag_1[i] := 0$ fi;
```

In line 1, process i (where $i \neq n$) first indicates that it is contending for the critical section by setting $Flag_1[i]$ to 1. Then, in the first repeat loop (lines 3–8) it finds out how many processes have higher priority than itself.

> A process k has higher priority than process i, if its second flag bit $Flag_2[k]$ is set to 1, or if $k < i$ and $Flag_1[k] = 1$.

If fewer than ℓ processes have higher priority, i exits the repeat loop (line 8). Otherwise, process i waits by spinning in the inner repeat loop (lines 3–8), until fewer than ℓ processes have higher priority. Once it exits the inner loop it sets its second flag, $Flag_2[i]$, to 1 (for $i \neq 1$). Then, again, it finds out how many processes have higher priority than itself. If fewer than ℓ processes have higher priority, process i exits the outer repeat loop (line 15) and can safely enter its critical section. Otherwise, the process sets its second flag bit back to 0, and goes back to wait in the inner repeat loop (lines 3–8). In the exit code a process simply sets its flag bits to 0.

3.2 Correctness Proof

We are assuming in this section that the registers are non-atomic. Thus, the individual read or write operations of different processes may overlap in time. A detailed model of non-atomic operations appears in [15]. In our case, to simplify the presentation, we only slightly modify the model presented in Subsection 2.1 as follows. With each read or write *operation* we associate two events, one event which marks the beginning of the operation and a corresponding event which marks the end of the operation. As before, a run is a sequence of events omitting all the states except the initial state. Thus, for an operation *op*, the pair of events *begin.op* and the corresponding *end.op* which appear in run x, mark the duration of *op* in x. Two operations op_1 and op_2 *overlap* in run x, if either *begin.op*$_2$ appears in x after *begin.op*$_1$ and before the corresponding *end.op*$_1$, or vice versa. As already mentioned, when a read operation of a non-atomic register overlaps a write operation into it, the value returned to the reader is arbitrary.

Theorem 3.2 *The two-bits algorithm satisfies ℓ-exclusion.*

Proof: Assume to the contrary that the algorithm does not satisfy ℓ-exclusion. This means that there is some finite run x and $\ell + 1$ processes, denoted $i_1, ..., i_{\ell+1}$, such that all these $\ell + 1$ processes are in their critical sections at the end of x. For every process $i \in \{i_1, ..., i_{\ell+1}\}$ such that $i \neq 1$, let x^i be the shortest prefix of x such that for every z, $x^i \leq z \leq x$, $Flag_2[i] = 1$ at the end of z. If $1 \in \{i_1, ..., i_{\ell+1}\}$, let x^1 be the shortest prefix of x such that for every z, $x^1 \leq z \leq x$, $Flag_1[1] = 1$ at the end of z. That is, in the last (end operation) event of x^i, process $i \neq 1$ sets $Flag_2[i]$ to 1 (line 9) for the last time before entering its critical section. If $1 \in \{i_1, ..., i_{\ell+1}\}$ then in the last (end operation) event of x^1, process 1 sets $Flag_1[i]$ to 1 (line 1) for the last time before entering its critical section.

Clearly x^i is shorter than x. Let us assume without loss of generality that $x^{i_1} \leq x^{i_2} \leq ... \leq x^{i_{\ell+1}}$. This implies that for every z, $x^{i_{\ell+1}} \leq z \leq x$, $Flag_2[i] = 1$ at the end of z for every $i \in \{i_1, ..., i_{\ell+1}\}$ where $i \neq 1$, and $Flag_1[1] = 1$ at the end of z if $1 \in \{i_1, ..., i_{\ell+1}\}$. This means that before process $i_{\ell+1}$ has finished changing $Flag_2[i_{\ell+1}]$ to 1 (or $Flag_1[1]$ to 1 in case $i_{\ell+1}$ is process 1), the values of all the flag bits of at least ℓ other processes are *already* set to 1 and thereafter stay continuously 1. Thus, before it enters its critical section, process $i_{\ell+1}$ will find that the values of the flag bits of ℓ (or more) other processes are 1 (lines 11–14). That is, it will find out that at least ℓ other processes have higher priority. We emphasize that, as explained above, during the time that $i_{\ell+1}$ reads the bits of these ℓ processes, these bits are not written to and thus, although the flag bits are non-atomic, $i_{\ell+1}$ will find out that they are set to 1. However, from the algorithm, a process, after changing its second flag bit to 1 (first bit in case of process 1) can *not* enter its critical section as long as it finds that ℓ or more other processes have higher priority. This contradict the fact that $i_{\ell+1}$ has succeeded to enter its critical section. ∎

Theorem 3.3 *The two-bits algorithm is ℓ–deadlock-free.*

Proof: We say that a process p *fails* in an infinite run x^∞, if there exists a finite prefix x of x^∞ such that p is not in its remainder at the end of x and $(x^\infty - x)$ does not involve p. Assume to the contrary that the algorithm is not ℓ-deadlock-free. This means that there is an infinite run, denoted x^∞, in which: at most $\ell - 1$ processes have failed, at least one non-faulty process is trying to enter its critical section, and all non-failed processes are unable to enter their critical sections from some point on. Let F, where $|F| \leq \ell - 1$, denote the set of all faulty processes in x^∞.

Thus, there is a finite prefix x of x^∞ where: (1) each process $p \notin F$ is either in its remainder or in its entry code at the end of x, (2) there is at least one process $p \notin F$ which is in its entry code at the end of x, and (3) no process changes to another region in any extension y of x where $y < x^\infty$. Let m be the number of processes not in F that are in their entry codes at the end of x, and let $\ell' = \min\{\ell, m\}$. In the following, we only consider runs which are prefixes of x^∞. Thus, for example, by "extension of x" we mean "extension of x which is a prefix of x^∞".

Let us denote by Q the set of all non-faulty processes that manage to *exit* the inner repeat loop (i.e, the repeat loop in lines 3–8) infinitely often in $(x^\infty - x)$. We prove that Q is an empty set. Assume to the contrary that $|Q| > 0$. Let p be the process with the largest identifier in Q. First, we observe that since no process changes to another region in any extension of x, in any extension of x, no process ever sets its first flag bit (i.e, $Flag_1[*]$) back to 0. Also, the second flag bit of a process (i.e, $Flag_2[*]$) is set to 1 only if its first flag bit is set to 1. Thus, since p has the largest identifier in Q, once p finds that there are at least ℓ processes with higher priority than itself, in *some* extension x' of x, p must also find that there are at least ℓ processes with higher priority than itself, in *any* extension of x' (which is a prefix of x^∞). (Recall that, a process k has higher priority than process i, if its second flag bit $Flag_2[k]$ is set to 1, or if $k < i$ and $Flag_1[k] = 1$.)

Thus, because of the similarity in the way the counting (of processes with higher priority) is done in the two *for* statements (lines 5–7 and lines 11–13), if p fails the test in line 15 of the outer repeat loop then p will also later fail the test in line 8 of the inner repeat loop. Thus, p will not be able to exit the inner repeat loop, a contradiction. We conclude that Q is an empty set, which implies that there is an extension y of x such that in $(x^\infty - y)$ all the events happen in the inner loop. Intuitively, this means that all the non-faulty processes (in the entry code) are stuck forever in the inner repeat loop.

Thus, in any extension of y of x, the first flag bits (i.e, $\mathit{Flag}_1[*]$) of the non-faulty processes in the entry code are set to 1, and all their other bits are set to 0. Let k be the non-faulty process with the smallest identifier, among all the non-faulty processes which are at the inner repeat loop in run y. When process k executes the for loop (lines 5–7), it must find that the (non-atomic) second flag bits (i.e., $\mathit{Flag}_2[*]$) of all the non-faulty processes in the range $k+1$ through n are 0. Hence, since there are at most $\ell - 1$ faulty processes, k must eventually exit the inner repeat loop, and later k must exit the outer repeat loop, and move into its critical section. This contradicts the assumption that no process changes its region in any extension of x. Thus, Algorithm 2 is ℓ-deadlock-free. ∎

3.3 The (ℓ, k)-exclusion Problem

The (ℓ, k)-exclusion problem is a simple generalization of the ℓ-exclusion problem. The problem is to write the code for the *entry code* and the *exit code* in such a way that the ℓ-exclusion and the k-deadlock-freedom requirements are satisfied, where $k \leq \ell$. The (ℓ, ℓ)-exclusion problem is the familiar ℓ-exclusion problem, and the $(1, 1)$-exclusion problem is the familiar mutual exclusion problem. It is easy to see that, for any $k \leq \ell$, a (k, k)-exclusion algorithm or an (ℓ, ℓ)-exclusion algorithm is also an (ℓ, k)-exclusion algorithm.

Theorem 3.4 *There is an (ℓ, k)-exclusion algorithm for n processes that uses,*

1. $2(n - \ell + k) - 2$ *non-atomic single-writer bits, for $n > \ell \geq k \geq 2$.*
2. $n - \ell + 1$ *non-atomic single-writer bits, for $n > \ell \geq 1$ and $k = 1$.*

Proof: To solve the (ℓ, k)-exclusion problem for n processes, we let the first $\ell - k$ processes enter and exit their critical sections whenever they want (no synchronization is needed) and use an algorithm for k-exclusion for the rest $n - (\ell - k)$ processes. It is easy to see that such a construction of an (ℓ, k)-exclusion algorithm satisfies the ℓ-exclusion and the k-deadlock-freedom requirements. When $k \geq 2$ we use the Two-bits k-exclusion algorithm (from Subsection 3.1), which uses $2(n - \ell + k) - 2$ non-atomic single-writer bits, for $n - \ell + k$ processes. When $k = 1$ we use the One-bit mutual exclusion algorithm (from [6, 9, 16]), which uses $n - \ell + 1$ non-atomic single-writer bits, for $n - \ell + 1$ processes. The result follows. ∎

4 Weak ℓ-exclusion

A *weak ℓ*-exclusion algorithm is an algorithm that satisfies (1) ℓ-exclusion, (2) 1-deadlock-freedom, and (3) weak ℓ-deadlock-freedom. Recall that weak ℓ-deadlock-freedom requires that: if strictly fewer than ℓ processes fail, at least one non-faulty process is trying to enter its critical section, and at least $n - \ell$ processes are in their remainders, then some non-faulty process eventually enters its critical section, provided that no process leaves its remainder in the meantime. For $\ell = 1$, a weak 1-exclusion algorithm is a mutual exclusion algorithm. Next we show that the tight bound for mutual exclusion [7, 9], of one bit per process, holds for weak ℓ-exclusion.

Theorem 4.1 *There is a weak ℓ-exclusion algorithm for n processes which uses one single-writer non-atomic bit per process, for $\ell \geq 1$.*

We present below a space optimal algorithm which generalizes the known One-bit mutual exclusion algorithm [6, 9, 16].

4.1 The Algorithm

There may be up to n processes potentially contending to enter their critical sections, each has a unique identifier from the set $\{1, ..., n\}$. The algorithm makes use of a shared array $Flag$, where, for every $1 \leq i \leq n$, all the processes can read the boolean registers $Flag[i]$, but only process i can write $Flag[i]$. ℓ is used as a constant. Initially all entries of the $Flag$ array are 0; the initial values of all the local variables are immaterial.

ALGORITHM 2: `process` $i \in \{1, ..., n\}$ `program.`

Shared: $Flag[1..n]$: array of non-atomic bits, initially all entries are 0.
Local: $lflag[1..n]$: array of bits; $counter$, j: integer ranges over $\{0, ..., n\}$.

```
1   $counter := 0$;
2   **repeat**
3       **if** $counter < \ell$ **then** $Flag[i] := 1$ **fi**;
4       $counter := 0$;
5       **for** $j := 1$ **to** $i - 1$ **do** $lflag[j] := Flag[j]$; $counter := counter + lflag[j]$ **od**;
6       **if** $counter \geq \ell$ and $Flag[i] = 1$ **then** $Flag[i] := 0$ **fi**;
7   **until** $Flag[i] = 1$;
8   **for** $j := i + 1$ **to** $n$ **do** $lflag[j] := Flag[j]$; $counter := counter + lflag[j]$ **od**;
9   **while** $counter \geq \ell$ **do**
10      **for** $j := 1$ **to** $n$ **do**
11          **if** $(Flag[j] = 0)$ and $(lflag[j] = 1)$
12          **then** $lflag[j] := 0$; $counter := counter - 1$ **fi**
13      **od**
14  **od**
15  *critical section*;
16  $Flag[i] := 0$;
```

In lines 1–7, process i first indicates that it is contending for the critical section by setting its flag bit to 1 (line 3), and then it tries to read the flag bits of all the processes which have identifiers smaller than itself. If fewer than ℓ of these bits are 0, i exits the repeat loop (line 7). Otherwise, i sets its flag bit to 0, waits until the values of fewer than ℓ of the flag bits of processes which have identifiers smaller than itself are 0 and starts all over again. In line 8, i reads the flag bits of all the processes which have identifiers greater than itself and remembers their values. Then, in the while loop in lines 9–14, it continuously reads the n flag bits, and it exits the loop only when it finds that at least $n - \ell$ of the flag bits have been 0 at least once since it has set its flag bit to 1. At that point it can safely enter its critical section.

4.2 Correctness Proof

We are assuming in this section that the registers are non-atomic. Thus, the individual read or write operations of different processes may overlap in time. In Subsection 3.2, it is explained how such a non-atomic behaviour is modeled.

Theorem 4.2 *Algorithm 2 satisfies ℓ-exclusion.*

Proof: Assume to the contrary that the algorithm does not satisfy ℓ-exclusion. This means that there is some finite run x and there are $\ell + 1$ processes, denoted $i_1, ..., i_{\ell+1}$, such that all these $\ell + 1$ processes are in their critical sections at the end of x. For every process $i \in \{i_1, ..., i_{\ell+1}\}$, let x^i be the shortest prefix of x such that for every z, $x^i \leq z \leq x$, $Flag[i] = 1$ at the end of z. That is, in the last event of x^i process i sets $Flag[i]$ to 1 (line 3) for the last time before entering its critical section. Clearly x^i is shorter than x. Let us assume without loss of generality that $x^{i_1} \leq x^{i_2} \leq ... \leq x^{i_{\ell+1}}$. This implies that for every z, $x^{i_{\ell+1}} \leq z \leq x$, $Flag[i] = 1$ at the end of z for every $i \in \{i_1, ..., i_{\ell+1}\}$. This means that before process $i_{\ell+1}$ completes changing its flag bit to 1, the values of the flag bits of at least ℓ other processes are *already* set to 1 and thereafter stay continuously 1. Thus, before it enters its critical section, process $i_{\ell+1}$ has to find that the values of the flag bits of ℓ (or more) other processes are 1. We emphasize that, as explained above, during the time that $i_{\ell+1}$ reads these ℓ bits they are not written and thus, although the flag bits are non-atomic, $i_{\ell+1}$ will find out that they are set to 1. However, from the algorithm, a process, after changing its flag bit to 1 cannot enter its critical section as long as it finds that the values of the flag bits of ℓ (or more) other processes are 1. This contradicts the fact that $i_{\ell+1}$ has succeeded to enter its critical section in x. ∎

Theorem 4.3 *Algorithm 2 is* 1*-deadlock-free.*

Proof: A process p *fails* in an infinite run x^∞, if there exists a finite prefix x of x^∞ such that p is not in its remainder at the end of x and $(x^\infty - x)$ does not involve p. An infinite run x^∞ is *fair* if no process fails in it. Assume to the contrary that the algorithm is not 1-deadlock-free. This means that there is an infinite fair run, say x^∞, and a finite prefix x of x^∞ where: (1) each process is either in its remainder or in its entry code at the end of x, (2) there is at least one process which is in its entry code at the end of x, and (3) no process changes to another region in any extension y of x where $y < x^\infty$. The third assumption implies also that no process is in its critical section at the end of x. Let m be the number of processes that are in their entry codes at the end of x, and let $\ell' = \min\{\ell, m\}$. In the following, we only consider runs which are prefixes of x^∞. Thus, for example, by "extension of x" we mean "extension of x which is a prefix of x^∞".

Let $P = \{k_1, ..., k_{\ell'}\}$ be the set of ℓ' processes with the smallest identifiers, among all the m processes which are in their entry codes at the end of x. Clearly $P \neq \emptyset$. The processes in P, are either executing the *while* loop (lines 9–14), or if not then there must be an extension of x in which they all reach the *while* loop. This must happen, since at any extension of x, $Flag[i]$ is 0 for all $i < k_1$, and hence every process in P will eventually, at some extension of x, will find out that fewer than ℓ of the non-atomic flag bits of the processes which have identifiers smaller than itself are set to 1, will exit the *repeat* loop, and reach the *while* loop.

We conclude that there is an extension of x in which all the processes in P are executing the *while* loop. Since it is assumed that no process leaves its entry code (in any extension of x), any process that reaches the *while* loop in lines 9–14, must stay there forever (in any extension of x). Thus, there is an extension of x, say y, in which: each process that is in its entry code, is either in the *while* loop, or it is busy-waiting in the *repeat* loop and its flag bit is forever 0. The reason why its flag bit is 0 is that if the flag bit is not forever 0 the process has to scan all the bits of processes smaller then itself and find that at most $\ell - 1$ of them are 0; but this is not possible because, in such a case $\min\{\ell, m\} = \ell$, the bits of processes in P are already set to 1 once they reach the *while* loop.

Thus, at y only the bits of the processes in the *while* loop are set to 1. Let k be the process with the largest identifier, among all the processes which are at the *while* loop, in run y. When process k

executes the *while* loop, it must find that the (non-atomic) flag bits of all the processes $k+1$ through n are 0, and hence it should move into its critical section. (Recall, that when process k exits the *repeat* loop, $counter < \ell$.) This contradicts the assumption that no process changes its region in any extension of x. Thus, Algorithm 2 is 1-deadlock-free. ▮

Theorem 4.4 *Algorithm 2 is ℓ-weak-deadlock-free.*

Proof: The proof is straightforward. Assume that there are at most ℓ processes which are not in their remainders. Then, when some process, say i, executes the *repeat* loop, its $counter$ will always be less than ℓ, and thus it will be able to immediately exit the loop. After executing the for loop at line 8, the value of the $counter$ will be at most $\ell - 1$ and after checking the condition of the *while* loop at line 9, process i will immediately continue into its critical section. ▮

We point out that for $\ell = 2$, the algorithm does not satisfy 2-deadlock-freedom. To see that, run process p_3 alone and let it crash in its critical section. Now run processes p_1 and p_2 until they set their flag bits to 1. From that point on, no process will be able to enter its critical section.

5 Discussion

For any ℓ and n, we provide a tight space bound on the number of single-writer bits required to solve ℓ-exclusion for n processes. It is easy to modify the two-bits algorithm (from Section 3), so that it uses a single 3-valued single-writer atomic register for $n-2$ of the processes and one bit per process for the remaining two processes. This, together with the result stated in Theorem 2.1, provides a tight space bound for the size and number of single-writer multi-valued registers required to solve ℓ-exclusion for n processes. We leave open the question of what is the bound for *multi-writer* registers.

There are two approaches for proving lower bounds and impossibility results, for distributed algorithms, by contradiction. The first is to show that a liveness property is violated. The second is to show that a safety property is violated. The first approach is usually used for cooperation problems such as consensus. For example, violating the liveness property that eventually all correct processes must decide [14, 17]. The second approach is usually used for contention problems such as mutual exclusion. For example, violating the safety property that two processes should never be in their critical sections at the same time [7, 9]. Interestingly, although ℓ-exclusion is a classical *contention* problem, we have used the *first* approach by violating the 2-deadlock-freedom liveness property.

The two algorithms are also resilient to the *failure by abortion* of any finite number of processes. By an abort-failure of process p, we mean that the program counter of p is set to point to the beginning of its remainder and that the values of all the single-writer bits of p are set to their initial (default) values. The process may then resume its execution, however, if a process keeps failing infinitely often, then it may prevent other processes from entering their critical sections.

Acknowledgement. I wish to thank the three anonymous referees and the associate editor Eric Ruppert for their constructive suggestions and corrections.

References

[1] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, 1994.

[2] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 262–272, October 1999.

[3] J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11, 1997.

[4] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 337–346, October 1987.

[5] A. Bar-Noy, D. Dolev, D. Koller, and D. Peleg. Fault-tolerant critical section management in asynchronous environments. *Information and Computation*, 95(1):1–20, November 1991.

[6] J. E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2):42–47, 1978.

[7] J.E. Burns and A.N. Lynch. Mutual exclusion using indivisible reads and writes. In *18th annual Allerton conference on communication, control and computing*, pages 833–842, 1980.

[8] J.E. Burns and G.L. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 145–158, August 1989.

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

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

[11] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: ℓ-exclusion as a test case. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 78–92, 1988.

[12] M.J. Fischer, N.A. Lynch, J.E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 234–254, October 1979.

[13] M.J. Fischer, N. A.Lynch, J.E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90–114, January 1989.

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

[15] L. Lamport. On Interprocess Communication, Parts I and II. *Distributed Computing*, 1, 2 (1986) 77–101.

[16] L. Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the ACM*, 33:327–348, 1986.

[17] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[18] G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).

[19] G. L. Peterson. Observations on ℓ-exclusion. In *28th annual allerton conference on communication, control and computing*, pages 568–577, October 1990.

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