Weak read/write Registers*

Gadi Taubenfeld

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

Abstract. In [14], Lamport has defined three classes of shared registers which support read and write operations, called —safe, regular and atomic—depending on their properties when several reads and/or writes are executed concurrently. We consider generalizations of Lamport's notions, called k-safe, k-regular and k-atomic. First, we provide constructions for implementing 1-atomic registers (the strongest type) in terms of k-safe registers (the weakest type). Then, we demonstrate how the constructions enable to easily and efficiently solve classical synchronization problems, such as mutual exclusion and ℓ-exclusion, using single-writer multi-reader k-safe bits, for any $k \geq 1$. We also explain how, by using k-registers, it is possible to provide some level of resiliency against memory reordering.

Keywords: k-safe, k-regular and k-atomic registers; shared memory, memory ordering, memory barriers, synchronization, mutual exclusion, ℓ-exclusion.

1 Introduction

It is common to assume that operations on the same memory location are atomic – they occur in some definite order. However, this assumption can be relaxed allowing the possibility of concurrent operation on the same memory location. In [14], Lamport has defined three classes of shared registers which support read and write operations, called —safe, regular and atomic—depending on their properties when several reads and/or writes are executed concurrently. Below we consider natural generalizations of Lamport's notions, motivate their use and investigate their properties. Unless otherwise stated, it is assumed that each register is a single-writer multi-reader register. Such a register can be written by one predefined process and can be read by all the processes. Let k be a positive integer.

– The weakest possibility is a *k-safe* register, in which it is assumed that a read not concurrent with any write obtains one of the k most recently written values. No assumption is made about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register. We consider the initial value as the first written value.

* A preliminary version of this paper (without Section 4 and Section 5) is to appear in the 14th international conf. on distributed computing and networking (ICDCN 2013), Mumbai, India, January 2013.

- The next stronger possibility is a k-*regular* register, in which it is assumed that a read not concurrent with any write obtains one of the k most recently written values. A read that overlaps a write obtains either the new value or one of the k most recently written values. That is, a read that overlaps any series of writes obtains either one of the values being written or one of the k most recently written values before the first of the writes.
- The final possibility is a k-*atomic* register, in which the reads and writes behave as if they occur in some definite order, and a read obtains one of the k most recently written values. In other words, for any execution, there is some way of totally ordering the overlapping reads and writes so that the value returned by the each read is one of the k most recently written values in the execution which has no overlapping. (Operations that do not overlap should take effect in their "real-time" order.)

We observe that Lamport's familiar notions of safe, regular and atomic registers are equivalent to the notions of 1-safe, 1-regular and 1-atomic registers, respectively. We will use the notion k-register as an abbreviation for k-safe, k-regular and k-atomic registers, when the exact type of a register is not important.

Our study is of both theoretical and practical interest. Various optimizations enable reordering memory references as it allows much better performance. When a correct operation depends on ordered memory references, memory barriers are used to prevent reordering. Memory barriers are required to enable good performance and scalability. The reason for that is the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access [17].

Without using memory barriers, as a result of reordering, a read from an atomic register may obtain some older value when compared to the value this read would return in the in order execution of the original program code. Suppose that in some setting where reordering is possible, a read may obtain, in the worst case, one of the 5 most recently written values when compared to the value it would return in the in order execution. In such a case, no harm done, if the program which uses 1-atomic registers, was designed in the first place to work correctly assuming that communication is done via 5-atomic registers.

Consider the following design strategy: Design your algorithms to be correct when k-atomic registers are used for some $k > 1$. Now replace the k-atomic registers with the stronger 1-atomic registers. In such algorithms the use of memory barriers may not be necessary in some cases, even when reordering is possible. Thus, there is a tradeoff between the number of memory barriers needed to ensure correctness and the type of k-registers used. Put another way, proving correctness w.r.t. k-registers while actually using 1-registers provides some level of resiliency against memory reordering. Finding the exact level of resiliency provided using such a design strategy, as a function of k, is an interesting research topic which is not covered in this paper.

Our results are about computability and complexity of using k-registers. We show that for any $k \geq 1$, k-safe registers and 1-atomic registers have the same computational power. More precisely, it is possible to wait-free implement multi-writer multi-reader multi-valued 1-atomic registers using single-writer single-reader k-safe bits, for any $k \geq 1$. We present simple and efficient constructions that enable to easily and efficiently

solve classical synchronization problems, such as mutual exclusion and ℓ-exclusion [20], using single-writer multi-reader k-safe registers, for any $k \geq 1$.

2 Preliminaries

We focus on an architecture in which n processes, denoted $p_1, ..., p_n$, communicate asynchronously via shared registers. A register can be either a single-writer single-reader (SWSR) register, a single-writer multi-reader (SWMR) register or a multi-writer multi-reader (MWMR) register. Unless explicitly stated, we assume that a register is a SWMR register. Asynchrony means that there is no assumption on the relative speeds of the processes. Processes may fail by crashing, which means that a failed process stops taking steps forever. We require that the constructions presented in this paper satisfy the wait-freedom progress condition. Wait-freedom guarantees that every process will always be able to complete its pending operations in a finite number of its own steps.

3 The constructions

We present two constructions of registers, by indicating how write operations and read operations are performed. The first construction implements a single-writer *multi-reader* multi-valued k-safe, k-regular or $(k+1)$-atomic register, denoted r, from single-writer *single-reader* multi-valued k-safe, k-regular or k-atomic registers, respectively.

Construction 1. *Let k be an arbitrary natural number, and let $r_1, ..., r_n$ be SWSR multi-valued k-registers, where each r_i $(i \in \{1, ..., n\})$ can be written by the same single process and read by process p_i. We construct a SWMR multi-valued k-register r as follows:*

- *The write operation $r := value$ is performed as follows:* **for** $i = 1$ **to** n **do** $r_i := value$*;*
- *The read operation of r by process p_i is performed by letting p_i read the value of r_i.*

The above construction is similar to Construction 1 from [14] which was designed for 1-registers. We prove the following theorem for the general case of k-registers.

Theorem 1. *The following claims are correct w.r.t. Construction 1, for any $k \geq 1$,*

1. *If $r_1, ..., r_n$ are SWSR k-safe registers or $r_1, ..., r_n$ are SWSR k-regular registers then r is a SWMR k-safe register or a SWMR k-regular register, respectively.*
2. *If $r_1, ..., r_n$ are SWSR k-atomic registers then r is a SWMR $(k+1)$-atomic register.*
3. *If $r_1, ..., r_n$ are SWSR k-atomic registers then r is not a SWMR k-atomic register.*

Proof. A read of r by process p_i that does not overlap a write of r, also does not overlap a write of r_i. If r_i is k-register (i.e., if r_i is k-safe, k-regular or k-atomic), then this read must obtain one of the k most recently written values into r. This is enough to show that if r_i is k-safe then r is k-safe. If a read of r_i by process p_i overlaps a write of r_i,

then it overlaps a write of the same value to r. In such a case, if r_i is k-regular then this read must obtain either the last value written or one of the k most recently written values into r_i (and hence into r). This implies that if r_i is k-regular then r is k-regular.

Now, assume that r_i is k-atomic, and that a read of r_i by process p_i overlaps a write of the value v into r. Then (1) if v was already written into r_i, this read must obtain either the value v or one of the $k-1$ most recently written values into r_i before v; or (2) if v was not written into r_i yet, this read must one of the k most recently written values into r_i. Since the linearization point of write of v into r might be before the linearization point of the read of v, in both cases above, the returned value is one of the $k+1$ most recently written values into r. This implies that if r_i is a SWSR k-atomic register then r is a SWMR $k+1$-atomic register.

Assume that $r_1, ..., r_n$ are k-registers. If a read of r by two different processes p_i and p_j both overlap the same write of value v into r, it is possible for p_i to get the new value v and for p_j to get the kth written value into r before the value v was written. This is possible even in the case where the read by p_i precedes the read by p_j. This possibility is not allowed by a k-atomic register. Thus, r is not a k-atomic register. □

The second construction implements a SWMR multi-valued 1-safe or 1-regular register from SWMR multi-valued k-safe or k-regular registers, respectively.

Construction 2. *Let k be an arbitrary natural number, and let r' be a SWMR multi-valued k-registers. We construct a SWMR multi-valued 1-register r as follows:*

- *The write operation $r := value$ is performed as follows:* **for** $i = 1$ **to** k **do** $r' := value$*;*
- *The read operation of r by process p is performed by letting p read the value of r'.*

Theorem 2. *The following claims are correct w.r.t. Construction 2, for any $k \geq 1$,*

1. *If r' is a SWMR k-safe register or a SWMR k-regular register then r is a SWMR 1-safe register or a SWMR 1-regular register, respectively.*
2. *If r' is a SWMR k-atomic register then r is not a SWMR 1-atomic register.*

Proof. A read of r by process p that does not overlap a write of r, also does not overlap any of the latest k writes of r'. Thus, all the k most recently written values into r' are identical and equal the most recent value written into r'. Since r' is k-register, in the case of no overlap, a read of r must obtain one of the k most recently written values into r', and thus it must obtain the most recent value written into r'. This is enough to show that if r' is k-safe then r is 1-safe.

If a read of r' by process p overlaps a write of r', then it overlaps a write of the same value to r. In such a case, if r' is k-regular then this read must obtain either the new value or one of the k most recently written values into r' (and hence into r). However, since each value is written k times, each of the k most recently written values equals either the new value or the most recent value written before the new value. This implies that if r' is k-regular then r is 1-regular.

We have assumed that r' is a k-register. Thus, during a write of r, the k most recently written values into r' equals either the new value or the most recent value written before the new value. If a read of r by two different processes p_i and p_j both overlap the same

write of value v into r, it is possible for p_i to get the new value v and p_j the old value. This is possible even in the case where the read by p_i precedes the read by p_j. This possibility is not allowed by a 1-atomic register. Thus, r is not a 1-atomic register. □

We notice that Construction 2 can be used for implementing a 2-atomic register from k-atomic registers. It would be interesting to find a similar simple and efficient construction also for implementing 1-atomic register from k-atomic registers.

Theorem 3. *It is possible to construct a MWMR 1-atomic register using SWSR k-safe bits.*

Proof. It follows immediately from Construction 1 and Construction 2 that it is possible to implement a SWMR 1-safe bit using SWSR k-safe bits. A well known result is that it is possible to implement a MWMR multi-valued 1-atomic register from SWMR 1-safe bits (see Chapter 4 of [10]). The result follows. □

The known constructions of a MWMR multi-valued 1-atomic register from SWMR 1-safe bits, are complicated and are not practically useful for transforming algorithms that use strong type of registers into algorithms that use weak type of registers.

4 Algorithms using k-safe bits

There are several classical synchronization algorithms that only use SWMR 1-safe registers, for interprocess communication [20]. Using Construction 2, such algorithms can be easily and efficiently modified to use only k-safe registers, for any $k \geq 1$. Below we demonstrate how this idea is used for solving the mutual problem and the ℓ-exclusion problem, using SWMR k-safe registers, for any $k \geq 1$.

4.1 Mutual Exclusion

The *mutual exclusion* problem, which was first introduced by Edsger W. Dijkstra in 1965, is the guarantee of mutually exclusive access to a single shared resource when there are several competing processes [6]. 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. The problem is of great significance, since it lies at the heart of many interprocess synchronization problems.

The mutual exclusion problem. The problem is formally defined as follows: 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*.

A process starts by executing the remainder code. At some point the process might need to execute some code in its critical section. In order to access its critical section a process has to go through an entry code which guarantees that while it is executing its critical section, no other process is allowed to execute its critical section. In addition, once a process finishes its critical section, the process executes its exit code in which

it notifies other processes that it is no longer in its critical section. After executing the exit code the process returns to the remainder.

The Mutual exclusion problem is to write the code for the *entry code* and the *exit code* in such a way that the following two basic requirements are satisfied.

Mutual exclusion: No two processes are in their critical sections at the same time.

Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.

The deadlock-freedom property guarantees that the system as a whole can always continue to make progress. However deadlock-freedom may still allow "starvation" of individual processes. That is, a process that is trying to enter its critical section, may never get to enter its critical section, and wait forever in its entry code. It is possible to define stronger requirements, which do not allow starvation.

A mutual exclusion algorithm using k-safe bits. The One-bit mutual exclusion algorithm, which uses n safe bits, was developed independently by Burns [3] (also appeared in [5]), and by Lamport [13]. We present below a space optimal mutual exclusion algorithm using only k-safe shared bits, which is a transformation of the One-bit algorithm using Construction 2. The correctness of the algorithms follows from that of the original One-bit algorithm and the properties of Construction 2 as expressed in part 1 of Theorem 2.

It is assumed that there may be up to n processes potentially contending to enter the critical section, each has a unique identifier from the set {1,...,n}. The algorithm makes use of an array b of k-safe bits, where, for every $1 \leq i \leq n$, all the processes can read the bits $b[i]$, but only process i can write $b[i]$. We will use the statement **await** *condition* as an abbreviation for: **while** $\neg$*condition* **do** *skip*. (The symbol $\neg$ means negation.) k is used as a constant. The places where the code of the original One-bits algorithm was altered appear in boxes. The algorithm is defined formally below.

THE ONE-BIT ALGORITHM USING k-SAFE BITS: `process i's program.`

Shared: $b[1..n]$: array of SWMR k-safe bits, initially all entries are 0.
Local: j: integer ranges over $\{0, ..., n\}$; inc: integer ranges over $\{1, ..., k\}$.

1 **repeat**
2 $\boxed{\textbf{for } inc := 1 \textbf{ to } k \textbf{ do } b[i] := 1 \textbf{ od;}}$ $j := 1$;
3 **while** ($b[i] = 1$ and ($j < i$) **do**
4 **if** $b[j] = 1$ **then** $\boxed{\textbf{for } inc := 1 \textbf{ to } k \textbf{ do } b[i] := 0 \textbf{ od;}}$ **await** $b[j] = 0$ **fi**;
5 $j := j + 1$
6 **od**
7 **until** $b[i] = 1$;
8 **for** $j := i + 1$ **to** n **do await** $b[j] = 0$ **od**;
9 *critical section*;
10 $\boxed{\textbf{for } inc := 1 \textbf{ to } k \textbf{ do } b[i] := 0 \textbf{ od;}}$

In lines 1–7, a process first indicates that it is contending for the critical section by setting its bit to 1 (line 2), and then it tries to read the bits of all the processes which have identifiers smaller than itself. If all these bits are 0 (while its bit is 1) the process exits the loop. Otherwise, the process sets its bit to 0, waits until the bit it is waiting on become 0 and starts all over again. In line 8, the process reads the bits of all the processes which have identifiers greater than itself. It exits this loop when it finds that each of the bits is 0 at least once since it entered this for-loop. Then it can safely enter the critical section.

4.2 ℓ-Exclusion

In [21], it is shown that, for any $\ell \geq 2$ and $n > \ell$, $2n-2$ single-writer bits are necessary and sufficient for solving ℓ-exclusion for n processes. We present below a space optimal ℓ-exclusion algorithm using only k-safe shared bits, which is a transformation of the Two-bits algorithm from [21] (which uses safe bits) using Construction 2. The correctness of the algorithms follows from that of the original algorithm and the properties of Construction 2 as expressed in part 1 of Theorem 2.

The ℓ-exclusion problem. To illustrate the ℓ-exclusion problem, which is a natural generalization of the mutual exclusion problem, consider the case of buying a ticket for a bus ride. Here a resource is a seat on the bus, and the parameter ℓ is the number of 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.

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.

ℓ-deadlock-freedom: If strictly fewer than ℓ processes fail (are delayed forever) 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".

An ℓ-exclusion algorithm using k-safe bits. The 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 SWMR k-safe bits, called $Flag_1[i]$ and $Flag_2[i]$. For process 1 the algorithm requires one SWMR k-safe bit, called $Flag_1[1]$, and for process n the algorithm requires one SWMR k-safe bit, called $Flag_2[n]$. In addition three local

variables, called $counter$, j and *inc*, are used for each process. ℓ and k are used as constants. The places where the code of the original Two-bits algorithm was altered appear in boxes.

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

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

```
1   if i ≠ n then [for inc := 1 to k do Flag1[i] := 1 od] fi;
2   repeat
3     repeat
4       counter := 0;
5       for j := 1 to n do
6         if (j < i and Flag1[j] = 1) or (Flag2[j] = 1)
7         then counter := counter + 1 fi od
8     until counter < ℓ;
9     if i ≠ 1 then [for inc := 1 to k do Flag2[i] := 1 od] fi;
10    counter := 0;
11    for j := 1 to n do
12      if (j < i and Flag1[j] = 1) or (j ≠ i and Flag2[j] = 1)
13      then counter := counter + 1 fi od
14    if counter ≥ ℓ then if i ≠ 1 then [for inc := 1 to k do Flag2[i] := 0 od] fi fi
15  until counter < ℓ;
16  critical section;
17  if i ≠ 1 then [for inc := 1 to k do Flag2[i] := 0 od] fi;
18  if i ≠ n then [for inc := 1 to k do Flag1[i] := 0 od] fi;
```

In lines 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 less 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 3–87), until less 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 less 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 go back to wait in the inner repeat loop (lines 3–8). In the exit code a process simply sets its flag bits to 0.

5 Related Work

A formalism for reasoning about concurrent systems which does not assume that read and write are atomic operations is developed in [12, 13]. This formalism has been further developed in [14], where it is used to specify several classes of interprocess communication mechanism and to prove correctness of algorithms for implementing them. In particular, a very weak form of (non-atomic) shared register, called *safe* register, is defined [14]. A safe register can be written and read concurrently, but read errors may occur during the writing of a shared register. Following Lamport's paper, many papers were published about implementing one type of shared register from another. In particular it is now known how to implement a MWMR atomic register from SWSR safe bits. A compilation of some of these results can be found in Chapter 4 of [10].

The relative power of various shared objects has been studied extensively in shared memory environments where processes may fail benignly, and where every operation is wait-free. In [11], a hierarchy of progressively stronger shared objects, is defined. Objects at each level are able to perform tasks which are impossible for objects at the lower levels. A discussion on how memory barriers are used to prevent reordering can be found in [15–17].

The mutual exclusion problem was first stated and solved for n processes by Dijkstra in [6]. 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 [4, 5], 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. The One-bit mutual exclusion algorithm, which uses n non-atomic (safe) bits and hence provides a tight space upper bound, was developed independently by Burns [3] (also appeared in [5]), and by Lamport [13]. The mutual exclusion algorithm using k-safe bits from Subsection 4.1, is based on the One-bit algorithm.

The ℓ-exclusion problem, which generalizes the mutual exclusion problem, was first defined and solved in [9, 8], for a model which supports read-modify-write registers. In [18], Peterson has proposed several ℓ-exclusion algorithms for solving the problem using atomic registers. Various other ℓ-exclusion algorithm are presented in [1, 7, 2]. Many known mutual exclusion and ℓ-exclusion algorithms are discussed in details in [19, 20].

In [21], it is shown that, for any $\ell \geq 2$ and $n > \ell$, $2n - 2$ single-writer bits are necessary and sufficient for solving ℓ-exclusion for n processes. The ℓ-exclusion algorithm using k-safe bits from Subsection 4.2, is based on the Two-bits algorithm from [21] which uses 1-safe bits.

6 Discussion

We have introduced the new notions of k-safe, k-regular and k-atomic registers, and showed how to implement 1-atomic registers (the strongest type) in terms of k-safe registers (the weakest type). We presented simple and efficient constructions that enabled us to solve classical synchronization problems, such as mutual exclusion and

ℓ-exclusion, using single-writer multi-reader k-safe bits, for any $k \geq 1$. On most modern microprocessors memory operations are not executed in the order specified by the program code. Memory reordering is used to fully utilize the different caches installed in such machines. Using k-registers provides some level of resiliency against memory reordering. The idea is to design an algorithm using k-registers and then (after proving its correctness w.r.t. the k-registers) to replace the k-registers with 1-registers. During run time, as a result of memory reordering, the 1-registers may exhibit a behavior of k-registers (w.r.t. the in order execution of the original program code), but that should not cause a problem as the algorithm was designed in advance to be correct when using k-registers. Exploring how the use of weak objects (like k-registers) can provide some level of resiliency against memory reordering and reduce the number of memory barriers required, is an interesting research topic.

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. J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11, 1997. .
3. J. E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2):42–47, (summer 1978).
4. 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, October 1980.
5. J.N. Burns and N.A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.
6. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
7. 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. .
8. 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.
9. 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.
10. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers. 508 pages, 2008.
11. M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.
12. L. Lamport. The mutual exclusion problem: Part I – a theory of interprocess communication. *Journal of the ACM*, 33:313–326, 1986.
13. L. Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the ACM*, 33:327–348, 1986.
14. L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.
15. P. E. McKenney. Memory ordering in modern microprocessors, part i. *Linux Journal*, 136, 2005. 2 page (Revised April 2009.).

16. P. E. McKenney. Memory ordering in modern microprocessors, part ii. *Linux Journal*, 137, 2005. 2 page (Revised April 2009.).
17. P. E. Mckenney. Memory barriers: a hardware view for software hackers, 2009.
18. G. L. Peterson. Observations on ℓ-exclusion. In *28th annual allerton conference on communication, control and computing*, pages 568–577, October 1990.
19. M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984
20. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall. ISBN 0-131-97259-6, 423 pages, 2006.
21. G. Taubenfeld. Tight space bounds for ℓ-exclusion. In *25th international symposium on distributed computing (DISC 2011)*, September 2011. *LNCS 6950* Springer Verlag 2011, 110–124.