Fair Synchronization*

Gadi Taubenfeld†

June 30, 2016

Abstract

Most published concurrent data structures which avoid locking do not provide any fairness guarantees. That is, they allow processes to access a data structure and complete their operations arbitrarily many times before some other trying process can complete a single operation. Such a behavior can be prevented by enforcing fairness. However, fairness requires waiting or helping. Helping techniques are often complex and memory consuming. Furthermore, it is known that it is not possible to automatically transform every data structure, which has a non-blocking implementation, into the corresponding data structure which in addition satisfies a very weak fairness requirement. Does it mean that for enforcing fairness it is best to use locks? The answer is negative.

We show that it is possible to automatically transfer any non-blocking or wait-free data structure into a similar data structure which satisfies a strong fairness requirement, without using locks and with limited waiting. The fairness we require is that no process can initiate and complete two operations on a given resource while some other process is kept waiting on the same resource. Our approach allows as many processes as possible to access a shared resource at the same time as long as fairness is preserved. To achieve this goal, we introduce and solve a *new* synchronization problem, called *fair synchronization*. Solving the new problem enables us to add fairness to existing implementations of concurrent data structures, and to transform any solution to the mutual exclusion problem into a fair solution.

Keywords: Synchronization, fairness, concurrent data structures, non-blocking, wait-freedom, locks, mutual exclusion.

*A preliminary version of the results presented in this paper, appeared in *proceedings of the 27th international symposium on distributed computing* (DISC 2013), Jerusalem, Israel, October 2013 [37].

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

1 Introduction

Motivation

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent data structures: a process accesses the data structure only inside a critical section code, within which the process is guaranteed exclusive access. However, using locks may degrade the performance of synchronized concurrent applications, as it enforces processes to wait for a lock to be released.

A promising approach is the design of data structures which avoid locking. Several progress conditions have been proposed for such data structures. Two of the most extensively studied conditions, in order of decreasing strength, are wait-freedom [18] and non-blocking [20]. Wait-freedom guarantees that every process will always be able to complete its pending operations in a finite number of its own steps. Non-blocking (which is sometimes also called lock-freedom) guarantees that some process will always be able to complete its pending operations in a finite number of its own steps.

Wait-free and non-blocking data structures are not required to provide fairness guarantees. That is, such data structures may allow processes to complete their operations arbitrarily many times before some other trying process can complete a single operation. Such a behavior may be prevented when fairness is required. However, fairness requires waiting or helping. Using helping techniques (without waiting) may impose too much overhead upon the implementation, and are often complex and memory consuming. Furthermore, it is known that using registers, it is not possible to automatically transform every data structure, which has a non-blocking implementation using registers, into the corresponding data structure which in addition satisfies a very weak fairness requirement, without using waiting [38].

Does it mean that for enforcing fairness it is best to use locks? The answer is negative. We show how any wait-free and any non-blocking implementation can be automatically transformed into an implementation which satisfies a very strong fairness requirement without using locks and with limited waiting.

We require that no beginning process can complete two operations on a given resource while some other process is kept waiting on the same resource. Our approach allows as many processes as possible to access a shared resource at the same time as long as fairness is preserved. To achieve this goal, we introduce and solve a new synchronization problem, called *fair synchronization*. Solving the fair synchronization problem enables us to add fairness to existing implementations of concurrent data structures, and to transform any solution to the mutual exclusion problem into a fair solution.

Fair Synchronization

The fair synchronization problem is to design an algorithm that guarantees fair access to a shared resource among a number of participating processes. Fair access means that no process can access a resource twice while some other process is kept waiting. There is no limit on the number of processes that can access a resource simultaneously. In fact, a desired property is that as many processes as possible will be able to access a resource at the same time as long as fairness is preserved.

It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, fair and exit. Furthermore, it is assumed that the entry section consists of two parts. The first part, which is called the *doorway*, is *fast wait-free*: its execution requires only a (very small) *constant* number of steps and hence always terminates; the second part is a *waiting* statement: it includes (at least one) loop with one or more statements. Like in the case of the doorway, the exit section is also required to be fast wait-free. A *waiting* process is a process that has finished its doorway code and reached the waiting part of its entry section. A *beginning* process is a process that is about to start executing its entry section.

A process is *enabled* to enter its fair section at some point in time, if sufficiently many steps of that process will carry it into the fair section, independently of the actions of the other processes. That is, an enabled process does not need to wait for an action by any other process in order to complete its entry section and enter its fair section, nor can an action by any other process prevent it from doing so.

The **fair synchronization problem** is to write the code for the entry and the exit sections in such a way that the following three basic requirements are satisfied.

- **Progress:** *In the absence of process failures and assuming that a process always leaves its fair section, if a process is trying to enter its fair section, then some process, not necessarily the same one, eventually enters its fair section.*

  The terms deadlock-freedom and livelock-freedom are used in the literature for the above progress condition, in the context of the mutual exclusion problem.

- **Fairness:** *A beginning process cannot execute its fair section twice before a waiting process completes executing its fair and exit sections once. Furthermore, no beginning process can become enabled before an already waiting process becomes enabled.*

  It is possible that a beginning process and a waiting process will become enabled at the same time. However, no beginning process can execute its fair section twice while some other process is kept waiting. The second part of the fairness requirement is called *first-in-first-enabled*. The term *first-in-first-out* (FIFO) fairness is used in the literature for a slightly stronger condition which guarantees that no beginning process can pass an already waiting process. That is, no beginning process can enter its fair section before an already waiting process does so.

- **Concurrency:** *All the waiting processes which are not enabled become enabled at the same time.*

  It follows from the *progress* and *fairness* requirements that *all* the waiting processes which are not enabled will eventually become enabled. The concurrency requirement guarantees that becoming enabled happens simultaneously, for all the waiting processes, and thus it guarantees that many processes will be able to access their fair sections at the same time as long as fairness is preserved. We notice that no lock implementation may satisfy the concurrency requirement.

Together the progress and fairness requirements imply that also the following property holds: In the absence of process failures and assuming that a process always leaves its fair

section, if a process is trying to enter its fair section, then this process, eventually enters its fair section. The term starvation-freedom is used in the literature for the above progress condition, in the context of the mutual exclusion problem.

The processes that have already passed through their doorway can be divided into two groups. The enabled processes and those that are not enabled. It is not possible to always have all the processes enabled due to the fairness requirement. All the enabled processes can immediately proceed to execute their fair sections. The waiting processes which are not enabled will eventually simultaneously become enabled, before or once the currently enabled processes exit their fair and exit sections.

The concurrency requirement is a special case of a set of conditions, recently introduced in [38], which are intended to capture the "amount of waiting" of processes in asynchronous concurrent algorithms. These new conditions can be described as follows. As already explained, a process is *enabled*, if by taking sufficiently many steps it will be able to complete its operation, independently of the actions of the other processes. A step is an *enabling* step, if after executing that step at least one process which was disabled becomes enabled. For a given $k \geq 0$, the *k-waiting* progress condition guarantees that every process that has a pending operation, will always become enabled once at most k enabling steps have been executed. The concurrency requirement is the same as requiring that the fair synchronization problem satisfies 1-waiting.

We observe that the stronger FIFO fairness requirement, the progress requirement and concurrency requirement cannot be mutually satisfied (Section 8). Fair Synchronization is a deceptive problem, and at first glance it seems very simple to solve. The only way to understand its tricky nature is by trying to solve it. We suggest the readers to try themselves to solve the problem, assuming that there are only three processes which communicate by reading and writing shared registers.

Contributions

Our model of computation consists of an asynchronous collection of n processes that (in most cases) communicate by reading and writing atomic registers. In a few cases, we will also define and consider stronger synchronization primitives. 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. Our contributions are as follows:

Fair synchronization. We define a new synchronization problem – called fair synchronization – for concurrent programming, show how it can be solved and demonstrate its importance. The problem is to design a highly concurrent algorithm that guarantees that no beginning process can access a resource twice while some other process is kept waiting on the same resource.

Algorithms. We present the first fair synchronization algorithm for n processes. The algorithm uses $n+1$ atomic registers: n 4-valued atomic registers plus one atomic bit. We also explain how to construct a fast and adaptive versions of the algorithm.

Fair data structures. We define the notion of a fair data structure and prove that by composing a fair synchronization algorithm and a non-blocking or a wait-free data structure, it is possible to construct the corresponding fair data structure.

Fair mutual exclusion algorithms. A fair mutual exclusion algorithm, in addition to satisfying the mutual exclusion and deadlock freedom requirements (Section 5), guarantees that no beginning process can access its critical section twice while some other process is kept waiting. We prove that by composing a fair synchronization algorithm and a deadlock-free mutual exclusion algorithm, it is possible to construct a fair mutual exclusion algorithm.

A space lower bound. We show that $n-1$ registers and conditional objects are necessary for solving the fair synchronization problem for n processes. Compare-and-swap and test-and-set are examples of conditional objects.

2 Preliminaries

Our model of computation consists of an asynchronous collection of n processes that communicate via shared objects. Asynchrony means that there is no assumption on the relative speeds of the processes. In most of the cases we considered, the shared objects are single-writer or multi-writer atomic registers.

A *single-writer* register can be written by one predefined process and can be read by all the processes. A *multi-writer* register can be written and 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. We will not consider non-atomic registers in this paper. In Section 6, we also consider communication via *conditional objects* such as compare-and-swap and test-and-set. These shared objects are formally defined in Section 6.

An event corresponds to an atomic step performed by a process. For example, 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. 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 describing our proofs, it would be more convenient to define a run as a sequence of events omitting all the states except the initial state.

A process executes a sequence of steps as defined by its algorithm. A process executes correctly its algorithm until it (possibly) crashes. After it has crashed a process executes no more steps. Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct*. In an asynchronous system there is no way to distinguish between a faulty process and a process that is very slow. A process is *active* (or participate) at a given run if it has taken at least one step in that run.

A fault-free model refers to a model where processes never fail. We will consider both the fault-free model, and models where processes may fail by crashing. In a model where participation is required, every process must eventually execute its code. However, a more interesting and practical situation is one in which participation is *not* required, as is more usually assumed when solving resource allocation problems or when designing concurrent data structures. For example, in the mutual exclusion problem a process can stay in the

reminder region forever and is not required to try to enter its critical section. In this paper we always assume that participation is not required.

3 The Fair Synchronization Algorithm

The fair synchronization algorithm, presented below, uses some key ideas from the Black-White Bakery algorithm presented in [34]. We use one (multi-writer multi-reader) atomic bit, called *group*. The first thing that process i does in its entry section is to read the value of the *group* bit, and to determine to which of the two groups (0 or 1) it should belong. This is done by setting i's single-writer register $state_i$ to the value read.

Once i chooses a group, it waits until its group has priority over the other group and then it enters its fair section. The order in which processes can enter their fair sections is defined as follows: If two processes belong to different groups, the process whose group, as recorded in its *state* register, is *different* from the value of the bit *group* is enabled and can enter its fair section, and the other process has to wait. If all the active processes belong to the same group then they can all enter their fair sections.

Next, we explain when the shared *group* bit is updated. The first thing that process i does when it leaves its fair section (i.e., its first step in its exit section) is to set the *group* bit to a value which is *different* from the value of its $state_i$ register. This way, i gives priority to waiting processes which belong to the same group that it belongs to.

Until the value of the *group* bit is first changed, all the active processes belong to the same group, say group 0. The first process to finish its fair section flips the value of the *group* bit and sets it to 1. Thereafter, the value read by all the new beginning processes is 1, until the group bit is modified again. Next, *all* the processes which belong to group 0 enter and then exit their fair sections possibly at the same time until there are no active processes which belong to group 0. Then all the processes from group 1 become enabled and are allowed to enter their fair sections, and when each one of them exits it sets to 0 the value of the *group* bit, which gives priority to the processes in group 1, and so on.

The following registers are used: (1) a single multi-writer atomic bit named *group*, (2) an array of single-writer atomic registers $state[1..n]$ which range over $\{0, 1, 2, 3\}$. To improve readability, we use below subscripts to index entries in an array. At any given time, process i can be in one of four possible states, as recorded in it single-writer register $state_i$. When $state_i = 3$, process i is not active, that is, it is in its remainder section. When $state_i = 2$, process i is active and (by reading *group*) tries to decide to which of the two groups, 0 or 1, it should belong. When $state_i = 1$, process i is active and belongs to group 1. When $state_i = 0$, process i is active and belongs to group 0.

The statement **await** *condition* is used as an abbreviation for **while** $\neg$*condition* **do** *skip*. The *break* statement, like in C, breaks out of the smallest enclosing *for* or *while* loop. Whenever two atomic registers appear in the same statement, two separate steps are required to execute this statement. Finally, to simplify the presentation, when the code for a fair synchronization algorithm is presented, only the entry and exit codes are described, and the remainder code and the infinite loop within which these codes reside are omitted. The algorithm is given below.

Algorithm 1. A Fair Synchronization Algorithm: `process` i`'s code` $(1 \leq i \leq n)$

Shared variables:

group: atomic bit; the initial value of the group bit is immaterial.
$state[1..n]$: array of atomic registers, which range over $\{0, 1, 2, 3\}$
Initially $\forall i : 1 \leq i \leq n : state_i = 3$ `/* processes are inactive */`

```
1   state_i := 2                                              /* begin doorway */
2   state_i := group                                /* choose group and end doorway */
3   for j = 1 to n do                                         /* begin waiting */
4       if (state_i ≠ group) then break fi                  /* process is enabled */
5       await state_j ≠ 2
6       if state_j = 1 − state_i                              /* different groups */
7       then await (state_j ≠ 1 − state_i) ∨ (state_i ≠ group) fi
8   od                                                        /* end waiting */
9   fair section
10  group := 1 − state_i                                      /* begin exit */
11  state_i := 3                                              /* end exit */
```

In line 1, process i indicates that it has started executing its doorway code. Then, in *two* atomic steps, it reads the value of *group* and assigns the value read to $state_i$ (line 2).

After passing its doorway, process i waits in the *for loop* (lines 3–8), until all the processes in the group to which it belongs are simultaneously enabled and then it enters its fair section. This happens when either, $(state_i \neq group)$, i.e. the value of the *group* bit points to the group which i does *not* belong to (line 4), or when all the waiting processes (including i) belong to the same group (line 7). Each one of the terms of the await statement (line 7) is evaluated separately. In case processes i and j belong to different groups (line 6), i waits until either (1) j is not competing any more or j has reentered its entry section, or (2) i has priority over j because $state_i$ is *different* than the value of the *group* bit.

In the exit code, i sets the *group* bit to a value which is different than the group to which it belongs (line 10), and changes its state to not active (line 11). We notice that the algorithm is also correct when we replace the order of lines 9 and 10, allowing process i to write the group bit immediately before it enters its fair section. The order of lines 10 and 11 is crucial for correctness.

We observe that a *none* beginning process, say p, may enter its fair section ahead of another waiting process, say q, twice: the first time if p is enabled on the other group, and the second time if p just happened to pass q which is waiting on the same group and enters its fair section first. We point out that omitting lines 1 and 5 will result in an incorrect solution. It is possible to replace each one of the 4-valued single-writer atomic registers, by three *separate* atomic bits. In Section 7, we present this variant of the algorithm which uses $3n + 1$ separate bits. Below we discuss two other interesting variants of Algorithm 1.

A fast fair synchronization algorithm: A fast algorithm is an algorithm whose time complexity, in the absence of contention, is constant [25]. Thus, a fair synchronization algorithm is fast if, in the absence of contention, the maximum number of times (i.e., steps) a process may need to access the shared memory in its entry and exit codes, is constant.

It is not difficult to make the fair synchronization algorithm (Algorithm 1) fast, using an additional atomic counter. The value of the counter is initially 0. The first step of a process is to atomically increment the counter by 1. After the process finishes executing its doorway (i.e., lines 1 and 2), it reads its value. If the returned value is 1, the process can safely enter its fair section, otherwise, the process continues to the waiting code (line 3). In the last step of its exit code the process decrements the counter by 1.

An adaptive fair synchronization algorithm: An adaptive algorithm is an algorithm whose time complexity is a function of the actual number of participating processes rather than a function of the total number of processes. In [2], a new object, called an *active set* was introduced, together with an implementation which is wait-free, adaptive and uses only atomic registers. The authors have shown how to transform the Bakery algorithm [22] into its corresponding adaptive version using the active set object. In [34], it was shown how to transform the Black White Bakery algorithm into its corresponding adaptive version using the same technique. It is rather simple to use the same transformation to make also the fair synchronization algorithm (Algorithm 1) adaptive.

Correctness proof

We prove below the correctness of the fair synchronization algorithm.

Theorem 3.1 *The fair synchronization algorithm for n processes (Algorithm 1) satisfies the progress, fairness and concurrency requirements, and uses $n+1$ atomic registers: n 4-valued single-writer atomic registers plus one multi-writer atomic bit. The total number of bits used is $2n+1$.*

The following lemma captures the effect of the group a process belongs to, on the order in which processes enter their fair sections.

Lemma 3.2 *For any two waiting processes i and j, if $state_i \neq group$ and $state_j = group$, then i must enter its fair section and complete its exit section before j can enter its fair section.*

Proof: A waiting process, say i, is *enabled* to enter its fair section only when one of the following two condition holds: (1) the value of $state_i \neq group$. In such a case, i will break out of the for loop after executing line 4; or (2) for all processes $j \neq i$, $state_j \neq 1 - state_i$. That is, no process belongs to a different group than the group i belongs to. In such a case, i will execute the loop n times and will exit. If none of these two conditions holds, i will eventually have to wait in line 7, until either the value of the group bit changes or the processes which belong to the other group change the values of their state registers.

Until the value of the *group* bit is first changed, all the active processes belong to the same group $v \in \{0,1\}$. Hence, as explained above, they are all enabled. The first process to finish its fair section flips the value of the *group* bit and sets it to $1-v$. Thereafter, the value of the group bit read by all the new beginning processes is $1-v$. As explained in the previous paragraph, none of these new beginning processes can become enabled until either, the value of the group bit changes again, or all the processes which belong to the group v complete their exit sections. Since all the processes which belong to the group v set the group bit to $1-v$ on their exit, the disabled processes will have to wait until all the enabled processes with state registers equal v exit.

Only then all the active processes belong to the same group $1 - v$, and hence will all become enabled. When they exit they change back to v the value of the *group* bit, and so on. As we can see in the above explanation, for any two waiting processes i and j, if $state_i \neq group$ and $state_j = group$, then i is enabled and j is disabled, and i and all the processes which belong to the same group as i will enter their fair sections and complete their fair and exit sections before j can enter its fair section. ∎

Proof of Theorem 3.1. The correctness of the claims about the number and size of the registers are obvious. Assume a beginning process i overtakes a waiting process j in entering its fair section. It follows from Lemma 3.2, that this can happen only if both i and j belong to the same group (i.e., $state_i = state_j$) at the time when i has completed executing line 2. On exit i (and possibly other processes) will set the value of the *group* bit to $1 - state_i$. Thereafter, by Lemma 3.2, the value of the group bit will not change (at least) until j completes its exit section. If i tries to enter its fair section again while j has not completed its exit section yet, then after passing through its doorway i will belong to a different group than j (i.e., $state_i \neq state_j$) and the value of the *group* bit will be the same as the value of $state_i$. Thus, by Lemma 3.2, i will not become enabled again until j - the process it has overtaken - completes its exit section and changes the value of its $state_j$ register. Thus, the algorithm satisfies *fairness*.

Next we assume towards a contradiction that the algorithm does not satisfy progress. Assuming that the algorithm does not satisfy progress means that there is an execution in which all the active processes are forced to remain in their entry sections forever. There are two possible cases: (1) the values of the state registers of all the active processes are the same, and (2) it is not the case that the values of the state registers of all the active processes are the same. In the first case, all the active processes are enabled and they all can proceed to their fair sections. In the later case, all the processes whose state register is different from the value of the group bit can proceed to their fair sections. In either case, some process can proceed. A contradiction. Thus, the algorithm satisfies *progress*.

We prove that the algorithm satisfies the concurrency requirement. As we have already explained in the proof of Lemma 3.2, a waiting process, say i, is *enabled* to enter its fair section only when one of the following two condition holds: (1) the value of $state_i \neq group$; in such a case, i will break out of the for loop after executing line 4; or (2) for all processes $j \neq i$, $state_j \neq 1 - state_i$. That is, no process belongs to a different group than the group i belongs to. In such a case, i will execute the loop n times and will exit. Thus, it follows that if two waiting processes, say i and j, are disabled then it must be the case that $state_i = state_j$. Lets assume that i becomes enabled. If i becomes enabled because the value of the group bit has changed then also j must become enabled for that reason. If i becomes enabled because no process belongs to a different group than the group i belongs to, then also j must become enabled for that reason. Thus, the algorithm satisfies *concurrency*. ∎

4 Fair Data Structures

In order to impose fairness on a concurrent data structure, concurrent accesses to a data structure can be synchronized using a fair synchronization algorithm: a process accesses the data structure only in its fair section. Any data structure can be easily made fair using such an approach, without using locks and with limited waiting.

We name a solution to the fair synchronization problem a (finger) *ring*.[1] Using a single *ring* to enforce fairness on a concurrent data structure, is an example of coarse-grained *fair* synchronization. In contrast, fine-grained *fair* synchronization enables to protect "small pieces" of a data structure, allowing several processes with *different* operations to access it completely independently. For example, in the case of adding fairness to an existing wait-free queue, it makes sense to use two rings: one for the enqueue operations and the other for the dequeue operations.

Coarse-grained fair synchronization is easier to program but might be less efficient compared to fine-grained fair synchronization. When using coarse-grained fair synchronization, operations that do not conflict may have to wait for one another, precluding disjoint-access parallelism. This can be resolved when using fine-grained fair synchronization.

4.1 Definitions

An implementation of each operation of a concurrent data structure is divided into two continuous sections of code: the doorway code and the body code. When a process invokes an operation it first executes the doorway code and then executes the body code. The *doorway* is *fast wait-free*: its execution requires executing only a *constant* number of instructions and hence always terminates.

A *beginning* process is a process that is about to start executing the doorway code of some operation. A process has *passed* its doorway, if it has finished the doorway code and reached the body code. A process is *enabled* while executing an operation on a given data structure, if by executing sufficiently many steps it will be able to complete its operation, independently of the actions of the other processes. That is, an enabled process does not need to wait for an action by any other process in order to complete its operation, nor can an action by any other process prevent it from doing so.

The problem of implementing a **fair data structure** is to write the doorway code and the body code in such a way that the following four requirements are satisfied,

- **Starvation-freedom (progress):** *In the absence of process failures, if a process is executing the doorway code or the body code, then this process, must eventually complete its operation.*

- **Fairness:** *No beginning process can complete an operation twice while some other process which has already passed the doorway has not completed its operation yet. Furthermore, no beginning process can become enabled before a process that has already passed its doorway becomes enabled.*

- **Concurrency:** *All the processes that have passed their doorway and are not enabled, become enabled at the same time.*

Recall the notion of the k-waiting condition (from [38]) as defined in the Section 1. The concurrency requirement above is the same as the 1-waiting condition. To keep things

[1] Many processes can simultaneously pass through the ring's hole, but the size of the ring may limit their number.

simple, we have not separated between the different types of operations a data structure may support. It is possible to refine the definition and, for example, require fairness only among operations of the same type.

We point out that it follows from a result published in [38], that by using registers, it is not possible to automatically transform every data structure, which has a non-blocking implementation using registers, into the corresponding data structure which satisfies both non-blocking and the above fairness requirement (or even a much weaker fairness requirement).

4.2 A composition theorem

By composing a fair synchronization algorithm and a non-blocking or a wait-free linearizable data structure, it is possible to construct a *fair* linearizable data structure. Linearizability is a consistency condition which means that although operations of concurrent processes may overlap, each operation should appear to take effect instantaneously, and operations that do not overlap should take effect in their "real-time" order [20]. The doorway code of the composed fair data structure is the doorway of the fair synchronization algorithm. The body is the waiting code of the fair synchronization algorithm followed by the code of the data structure, followed by the exit section.

Theorem 4.1 *Let A be a fair synchronization algorithm and let B be a non-blocking or a wait-free implementation of a data structure. Assume that the registers of A are different from the registers of B. Let C be an implementation of a data structure obtained by replacing the fair section of A with the implementation B. Then, C is an implementation of a fair data structure. Furthermore, if B is linearizable, then also C is linearizable.*

Proof: In the definition of a fair synchronization algorithm it is assumed that a process that is executing its fair section, always leaves its fair section. In the composition, we have considered B (i.e., the embedded data structure) as the fair section of the algorithm A (i.e, the fair synchronization algorithm). Thus, in order to prove that the composition satisfies the progress requirement, we first need to show that in any execution of C (i.e, of the composition) executing B always terminates. That is, any process that executes B after it has completed executing the entry code of A *must* eventually complete executing B. This is clearly the case when B is a wait-free data structure, we prove that it also the case when B is non-blocking.

Let p be a process that is executing the embedded data structure B. If p is the only process that is currently executing B then, since B is non-blocking, p must eventually complete executing B. Otherwise, some other process, say q, will complete executing B. By the fairness requirement of A, q will be able to bypass p at most one more time. (Notice that q might have not been a beginning process the first time it bypassed p and hence it might be able to bypass p once more.) The above argument about q is correct for any process that bypasses p. Thus, eventually either p will complete executing B or no process will be able to bypass p. In the later case, again since B is non-blocking, p will eventually complete executing B. This completes the proof that executing the embedded data structure B always terminates. This implies that, in the absence of process failures, algorithm A satisfies its progress requirement (deadlock-freedom) also within the composition.

It follows from the fact that in the composition A satisfies the progress and fairness requirements that, in the absence of process failures, every process that has started executing

its entry section of A eventually completes its entry section. Furthermore, we have already shown that, in the composition every process that executes B after it has completed executing the entry section of A *must* eventually complete executing B (regardless of whether B is wait-free or non-blocking). Since the exit section of A is wait-free, it follows that, in the absence of process failures, if a process is executing the doorway code or the body code (of C), then this process, must eventually complete its operation. Thus, C satisfies *starvation-freedom*.

The embedded code of B within C is preceded and hence protected by the entry section of A. From the fact that A satisfies fairness, it follows that no beginning process (of C) can execute B (which replaces the fair section of A) twice before a waiting process (that is, a process that has passed its doorway) completes B and the exit section of A once. Furthermore, no beginning process can become enabled before a process that has already passed its doorway. This implies that a beginning process (of C) cannot complete an operation twice while some other process which has already passed the doorway has not completed its operation yet. Furthermore, no beginning process can become enabled before a process that has already passed its doorway. Thus, C satisfies *fairness*.

We have already shown that, in the composition every process that executes B after it has completed executing the entry section of A *must* eventually complete executing B (regardless of whether B is wait-free or non-blocking) and then it must complete the wait-free exit section of A. This means that every process that starts executing B is enabled. The above observation together with the fact that A satisfies the concurrency requirement, implies that all the processes that have passed the doorway of C and are not enabled, become enabled at the same time. Thus, C satisfies *concurrency*.

Finally, from the fact that B is linearizable, it follows that C is linearizable. This is so, because for every operation of C, we can take its linearization point to be the same as the linearization point of the embedded operation of B. Thus, C satisfies *linearizability*. ∎

Using Theorem 4.1, it is now possible to implement new fair data structures from existing non-blocking or wait-free implementations of data structures.

5 Fair Mutual Exclusion

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical and exit. The entry section consists of two parts: the *doorway* which is *wait-free*, and the waiting part which includes one or more loops. Recall that a *waiting* process is a process that has finished its doorway code and reached the waiting part, and a *beginning* process is a process that is about to start executing its entry section. Like in the case of the doorway, the exit section is also required to be wait-free. It is assumed that processes do not fail, and that a process always leaves its critical section.

5.1 Definitions

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

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.*

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

Satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm. For an algorithm to be fair, satisfaction of an additional condition is required.

k-fairness: *A beginning process cannot execute its critical section $k+1$ times before a waiting process completes executing its critical and exit sections once.*

We notice that 1-fairness implies that no beginning process can execute its critical section twice while some other process is kept waiting. The term first-in-first-out (FIFO) is used for 0-bounded-waiting: no beginning process can pass an already waiting process. The term *linear-waiting* is used in the literature for the requirement that no (beginning or not) process can execute its critical section twice while some other process is kept waiting.

The **fair mutual exclusion problem** is to write the code for the entry and exit sections in such a way that the deadlock-freedom, mutual exclusion and 1-fairness requirements are satisfied. Solving the fair synchronization problem enables to transform *any* solution for the mutual exclusion problem into a fair solution.

5.2 A composition theorem

By composing a fair synchronization algorithm (FS) and a deadlock-free mutual exclusion algorithm (ME), it is possible to construct a *fair* mutual exclusion algorithm (FME). The entry section of the composed FME algorithm consists of the entry section of the FS algorithm followed by the entry section of the ME algorithm. The exit section of the FME algorithm consists of the exit section of the ME algorithm followed by the exit section of the FS algorithm. The doorway of the FME algorithm is the doorway of the FS algorithm.

Theorem 5.1 *Let A be a fair synchronization algorithm and let B be a deadlock-free mutual exclusion algorithm. Assume that the registers of A are different from the registers of B. Let C be the algorithm obtained by replacing the fair section of A with the algorithm B. That is, the code of C is* ***loop forever*** remainder code (of C)*;* entry code of A*;* entry code of B*;* critical section*;* exit code of B*;* exit code of A ***end loop****.*
Then, C is a fair mutual exclusion algorithm.

Proof: In the definition of a fair synchronization algorithm, it is assumed that a process that is executing its fair section always leaves its fair section. In the composition, we have considered B (i.e., the embedded mutual exclusion algorithm) as the fair section of the algorithm A (i.e, the fair synchronization algorithm). Thus, in order to prove that the composition satisfies the deadlock-freedom requirement, we first need to show that although B satisfies only deadlock-freedom, in any execution of C (i.e, of the composition) the entry code of B is terminating. That is, in the absence of process failures, any process that

executes the entry code of B after it has completed executing the entry code of A *must* eventually complete executing the entry code of B. (Recall that the critical section and the exit code of B are assumed to be terminating.)

Let p be a process that is executing its embedded entry section of B. If p is the only process that is currently executing the entry code of B then, since B is deadlock-free, p will eventually complete its entry code of B. Otherwise, some other process, say q, will complete its entry code of B. By the fairness requirement of A, q will be able to bypass p at most one more time. (Notice that q might have not been a beginning process the first time it bypassed p and hence it might be able to bypass p once more.) The above argument about q is correct for any process that bypasses p. Thus, eventually either p will enter its critical section or no process will be able to bypass p. In the later case, again since B is deadlock-free, p will eventually enter its critical section. This completes the proof that the embedded entry section of B is terminating. This implies that algorithm A satisfies its progress condition (deadlock-freedom) also within the composition.

Since, within the composition, A satisfies its progress condition and B satisfies deadlock-freedom, it follows immediately that C also satisfies *deadlock-freedom*. The critical section of C (which is essentially the embedded critical section of B) is preceded and hence protected by the entry section of B. The fact that B satisfies mutual exclusion means that no two processes are in their critical sections of B at the same time. Since the critical section of C is essentially the embedded critical section of B, it follows that no two processes are in their critical sections of C at the same time, which means that C satisfies *mutual exclusion*. Finally, from the fact that A satisfies fairness, it follows that no beginning process (of C) can execute B (which replaces the fair section of A) twice before a waiting process (that is, a process that has passed its doorway) completes B and the exit section of A once. This means that a beginning process (of C) cannot execute its critical section (of C) twice before a waiting process completes executing its critical and exit sections (of C) once. Thus, C satisfies *1-fairness*. ∎

Using Theorem 5.1, it is now possible to construct new interesting fair mutual exclusion algorithms. For example, the One-bit algorithm that was devised independently in [5, 6] and [24], is a deadlock-free mutual exclusion algorithm for n processes which uses n shared bits. The algorithm is especially interesting since it uses the minimum possible shared space in the case of deadlock-free mutual exclusion. By Theorem 5.1, using the fair synchronization algorithm from Section 3 which uses $2n + 1$ bits together with the One-bit algorithm which uses n bits, we can construct an elegant and simple fair mutual exclusion algorithm which uses $3n + 1$ bits.

Several techniques for designing FIFO mutual exclusion algorithms have been used in [21, 24, 26]. It is interesting to note that while the doorway of the above new fair mutual exclusion algorithm includes only three steps (accessing $state_i$ twice and *group* once), the doorway of the various FIFO mutual exclusion algorithms [21, 24, 26] is not fast wait-free as it takes at least n steps, where n is the number of processes. Next we use Theorem 5.1 for proving a space lower bound for the fair synchronization problem.

6 A Space Lower Bound for Fair Synchronization

In Section 3, we have shown that $n + 1$ atomic registers are sufficient for solving the fair synchronization problem for n processes. In this section we show that $n - 1$ registers

and conditional objects are necessary for solving the fair synchronization problem for n processes. A conditional operation is an operation that changes the value of an object only if the object has a particular value. A *conditional object* is an object that supports only conditional operations. Compare-and-swap and test-and-set are examples of conditional objects.

A compare-and-swap operation takes a register r, and two values: *new* and *old*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise r is left unchanged and the value *false* is returned. A compare-and-swap object is a register that supports a compare-and-swap operation. A test-and-set operation takes a registers r and a value *val*. If the value of r is 0 (the initial value) the value *val* is assigned to r, and 0 is returned, otherwise r is left unchanged and the value of r is returned. A test-and-set bit is an object that supports a reset operation (i.e., write 0) and a restricted test-and-set operation where the value of *val* can only be 1.

Theorem 6.1 *Any fair synchronization algorithm for n processes using only atomic registers and conditional objects must use at least $n-1$ atomic registers and conditional objects.*

Proof: A deadlock-free mutual exclusion algorithm using a single test-and-set bit is defined as follows. It uses a test-and-set bit called x. In its entry section, a process keeps on accessing x until, in one atomic step, it succeeds to change x from 0 to 1. Then, the process can safely enter its critical section. The exit section is simply to reset x to 0. By Theorem 5.1, it is possible to construct a fair mutual exclusion algorithm (FMX) by composing any fair synchronization algorithm and the above deadlock-free mutual exclusion algorithm.

A starvation-free mutual exclusion is an algorithm that satisfies the mutual exclusion requirement and guarantees that, in the absence of process failures, any process that tries to enter its critical section eventually enters its critical section. Clearly, any FMX algorithm is also a starvation-free mutual exclusion algorithm.

In [29], it is proven that any starvation-free mutual exclusion algorithm for n processes using only atomic registers and test-and-set bits must use at least n atomic registers and test-and-set bits. In [12] it is proven that any starvation-free mutual exclusion algorithm for n processes using only atomic registers and conditional objects must use at least n atomic registers and conditional objects. Since, a FMX algorithm is also a starvation-free mutual exclusion algorithm, the above lower bound holds also for FMX algorithms.

It follows from the two facts that (1) we can construct a FMX algorithm using any fair synchronization algorithm plus a single test-and-set bit, and that (2) any FMX algorithm for n processes using only atomic registers and conditional objects must use at least n atomic registers and conditional objects, and that any *fair synchronization* algorithm for n processes using only atomic registers and conditional objects must use at least $n-1$ atomic registers and conditional objects. ∎

7 Fair synchronization using 3n+1 separate atomic bits

We present a variant of the fair synchronization algorithm from Section 3 which uses $3n+1$ *separate* bits. That is, for each process i, the 4-valued register $state_i$ is replaced with three *separate* single-writer bits. The following registers are used: (1) a single multi-writer multi-reader atomic bit named *group*, (2) two boolean array $flag[1..n]$ and $choosing[1..n]$, and

(3) an array of atomic bits $mygroup[1..n]$. The algorithm is given below.

Algorithm 2. A Fair Synchronization Algorithm: `process` i`'s code` $(1 \leq i \leq n)$

Shared variables:

$group$: atomic bit
$flag[1..n]$, $choosing[1..n]$: boolean arrays
$mygroup[1..n]$: array of atomic bits
Initially $\forall i : 1 \leq i \leq n : flag_i =$ `false` $\wedge\ choosing_i =$ `false`,
the initial values of all the other variables are immaterial

```
1   choosing_i := true                                          /* begin doorway */
2   mygroup_i := group                                          /* choose group */
3   flag_i := true                                              /* i is active */
4   choosing_i := false                                         /* end doorway */
5   for j = 1 to n do                                           /* begin waiting */
6       if (mygroup_i ≠ group) then break fi
7       await choosing_j = false
8       if mygroup_j ≠ mygroup_i
9       then await (flag_j = false) ∨ (mygroup_i ≠ group) ∨
                   (mygroup_j = mygroup_i) fi
10  od                                                          /* end waiting */
11  fair section
12  group := 1 − mygroup_i                                      /* begin exit */
13  flag_i := false                                             /* end exit */
```

In line 1, process i indicates that it starts executing its doorway code. Then, in two separate atomic steps, it reads the value of *group* and assigns the value read to $mygroup_i$ (line 2). Then it indicates that it is active by setting its *flag* bit to true (line 3).

After passing the doorway, process i waits in the *for loop* (lines 5–10), until all the processes in the group to which it belongs are enabled and then it enters its fair section. This happens when either, $(mygroup_i \neq group)$ (i.e., the value of the *group* bit points to the group which i does *not* belong to) (line 6), or when all the waiting processes (including i) belong to the same group (line 9). We notice that each one of the three terms of the await (line 9) statement is evaluated separately. In case processes i and j belong to different groups (line 9), i waits until it notices that either (1) j is not competing any more, (2) i has priority over j because $mygroup_i$ is *different* than the value of the *group* bit, or (3) j has reentered its entry section. In the exit code (line 12), i sets the *group* bit to a value which is different than the group to which it belongs, and sets its flag bit to false (line 13). The correctness proof is similar to that of Algorithm 1, with minor changes.

We notice that the algorithm is also correct when we replace the order of lines 11 and 12, allowing process i to write the group bit immediately before it enters its fair section. Also, the order of lines 12 and 13 is crucial for correctness. The correctness proof of Algorithm 2 is similar to that of algorithm 1 from Section 3.

8 FIFO fairness

In the definition of the *fair synchronization problem* it is required that three basic requirements are satisfied. It is natural to ask why we have not used the stronger FIFO fairness requirement. Namely, that no beginning process can enter its fair section before an already waiting process does so. The answer is simple, it is impossible to solve the fair synchronization problem when using the stronger FIFO fairness requirement.

Theorem 8.1 *The FIFO fairness requirement, the progress requirement and concurrency requirement cannot be mutually satisfied.*

Proof: Assume to the contrary that they can be mutually satisfied. Let p_1, p_2, p_3 and p_4 be beginning processes. Run p_1, p_2, p_3 and p_4, each one alone and in that order, until each one of them completes its doorway code. By the FIFO fairness requirement, for $1 \leq i < j \leq 4$, p_j cannot enter its fair section before p_i does so. Thus, p_2, p_3, p_4 are not enabled (otherwise, we can let them enter their fair sections before p_1 does so). Now, let the processes continue until p_1 enters its fair section. The other three processes are still in their entry sections. Run p_1 until it finishes its exit code. Now run p_2, p_3 and p_4 until one of them becomes enabled (this must happen by the progress requirement). By the concurrency requirement, once one process is enabled all the three processes must be enabled. W.l.o.g. assume that the last event before they all become enabled is not by p_2 or p_3. Thus, p_2 and p_3 must still be in their entry sections. Now, run p_3 alone until it enters its fair section before p_2 does so. A contradiction. ∎

9 Related Work

Mutual exclusion locks were first introduced by Edsger W. Dijkstra in [8]. Since then, numerous implementations of locks have been proposed [31, 33]. Various other types of locks, like ℓ-exclusion locks [14, 13] and read/write locks [7], were considered in the literature. For each type of a lock it is *a priori* defined how many processes and/or which processes (i.e., a reader process or a writer process) cannot be in their critical sections at the same time. In the case of the fair synchronization problem no such a priori requirement exists. The fair synchronization algorithm, presented in Section 3, uses some key ideas from the mutual exclusion algorithm presented in [34].

Implementations of data structures which avoid locking have appeared in many papers, a few examples are [10, 15, 16, 28, 32, 39]. Several progress conditions have been proposed for data structures which avoid locking. The most extensively studied conditions are wait-freedom [18] and non-blocking [20]. Strategies that avoid locks are called lockless [17] or lock-free [27]. (In some papers, lock-free means non-blocking.) Consistency conditions for concurrent objects are linearizability [20] and sequential consistency [23]. A tutorial on memory consistency models can be found in [1].

In [38], it is shown that between the two extremes, *lock-based* algorithms, which involve "a lot of waiting", and *wait-free* algorithms, which are "free of locking and waiting", there is an interesting spectrum of different levels of waiting. New progress conditions, called k-waiting, for $k \geq 0$, which are intended to capture the "amount of waiting" of processes in asynchronous concurrent algorithms, were introduced. The concurrency requirement used

in the definition of the fair synchronization problem is the same as the 1-waiting condition. To illustrate the utility of the new k-waiting conditions, they have been used in [38] to derive new lower and upper bounds, and impossibility results for well-known basic problems such as consensus, election, renaming and mutual exclusion. Furthermore, the relation between waiting and fairness was explored.

In order to improve wait-free object implementations, in [3, 4], it is suggested to first protect a shared object by an ℓ-exclusion lock; processes that passed the ℓ-exclusion lock, rename themselves before accessing the object. This enables the usage of an object that was designed only for up to ℓ processes, rather than a less efficient object designed for n processes. The implementation uses strong synchronization primitives.

An algorithm is *obstruction-free* if it guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes hold still long enough (that is, in the absence of interference from other processes) [19]. Transformations that automatically convert any obstruction-free algorithm into a non-blocking or a wait-free algorithm are presented in [11, 35], for a model where it is assumed that there is a (possibly unknown) upper bound on memory access time.

Contention-sensitive data structures in which the overhead introduced by locking is eliminated in common cases, when there is no contention, or when processes with non-interfering operations access it concurrently, are introduced in [36]. Hybrid implementations of concurrent objects in which lock-based code and lock-free code are merged in the same implementation of a concurrent object, are discussed in [30].

In [9], the authors considered the fair synchronization problem (following the publication of [37]) in read/write asynchronous systems where any number of processes may crash. They have introduced a new failure detector and used it to solve the fair synchronization problem when processes may crash. They also showed that the proposed failure detector is optimal in the sense that the information on failures it provides to the processes can be extracted from any algorithm solving the fair synchronization problem in the presence of any number of process crash failures.

10 Discussion

We have presented fair synchronization - an algorithm consisting of an entry and exit sections, that encapsulate blocking and non-blocking object implementations and enforces fair access patterns of the encapsulated objects. Fairness means that no process may access the encapsulated object (be it a mutually-exclusive critical section or a non-blocking data-structure) twice while another process is waiting. A linear space lower bound has been obtained for the fair synchronization problem.

Wait-free algorithms are frequently criticized for sacrificing performance compared to non-blocking algorithms. When enforce fairness, it is better that the basic concurrent algorithm be an efficient non-blocking algorithms rather than a wait-free algorithm. Since many processes may enter their fair sections simultaneously, it is expected that using fair synchronization algorithms will not degrade the performance of concurrent applications as much as locks. However, as in the case of using locks, slow or stopped processes may prevent other processes from ever accessing their fair sections.

There are several interesting variants of the fair synchronization problem which can be

defined by strengthening or weakening the various requirements. For example, it is possible to require that a solution be able to withstand the slow-down or even the crash (fail by stopping) of up to $\ell - 1$ of processes. In that variant, the (stronger) progress condition is as follows: If strictly fewer than ℓ processes fail (are delayed forever) then if a process is trying to enter its fair section, then some process, not necessarily the same one, eventually enters its fair section. Solving the problem with such a strong progress requirement should be possible only by weakening the fairness requirement.

According to our definition of fairness, there is no overtaking. Such a requirement might be too strong. Allowing a process to access a shared resource for a constant number of times while another is spinning on it, is a reasonable weakening of the current definition. Some version of the two composition theorems would still hold for such weaker versions, and this might be closer to what happens in real life. Put another way, it is possible to replace the fairness requirement by k-fairness (as defined in Section 5) for some $k > 1$.

Like in the case of mutual exclusion, it would be interesting to solve the fair synchronization problem using synchronization primitives other than atomic registers, prove time complexity bounds, and find local spinning, symmetric, self stabilizing and fault-tolerant solutions.

Acknowledgements: I wish to thank the two anonymous referees for their constructive suggestions and corrections.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[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 k-exclusion to implement resilient, scalable shared objects. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 141–150, August 1994.

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

[5] 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.

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

[7] P.L. Courtois, F. Heyman, and D.L Parnas. Concurrent control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, 1971.

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

[9] C. Delporte-Gallet, H. Fauconnier, and M. Raynal. Fair synchronization in the presence of process crashes, and its weakest failure detector. In *Proceedings of the 33rd Intl Symposium on Reliable Distributed Systems (SRDS 14)*, pages 161–170, 2014.

[10] W. B. Easton. Process synchronization without long-term interlock. In *Proc. of the 3rd ACM symp. on Operating systems principles*, pages 95–100, 1971.

[11] E. F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *19th international symposium on distributed computing*, 2005. *LNCS 3724* Springer Verlag 2005, 78-92.

[12] F. E Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional synchronization primitives. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 80–87, 2004.

[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, 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.

[15] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 50–59, 2004.

[16] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *15th international symposium on distributed computing*, October 2001.

[17] T. E. Hart, P. E. McKenney, , and A. D. Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *Proc. of the 20th international Parallel and Distributed Processing Symp.*, 2006.

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

[19] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, page 522, 2003.

[20] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *toplas*, 12(3):463–492, 1990.

[21] H.P. Katseff. A new solution to the critical section problem. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 86–88, May 1978.

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

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 28(9):690–691, September 1979.

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

[25] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[26] E. A. Lycklama and V. Hadzilacos. A first-come-first-served mutual exclusion algorithm with small communication variables. *ACM Trans. on Programming Languages and Systems*, 13(4):558–576, 1991.

[27] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[28] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

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

[30] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer. ISBN 978-3-642-32027-9, 515 pages, 2013.

[31] M. Raynal. *Algorithms for mutual exclusion.* The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[32] H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *8th International Conference on Principles of Distributed Systems*, 2004.

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

[34] G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 56–70.

[35] G. Taubenfeld. Efficient transformations of obstruction-free algorithms into non-blocking algorithms. In *21st international symposium on distributed computing (DISC 2007)*, September 2007. *LNCS 4731* Springer Verlag 2007, 450–464.

[36] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *23rd international symposium on distributed computing (DISC 2009)*, September 2009. *LNCS 5805* Springer Verlag 2009, 157–171.

[37] G. Taubenfeld. Fair synchronization. In *27th international symposium on distributed computing*, October 2013. *LNCS 8205* Springer Verlag 2013, 179–193.

[38] G. Taubenfeld. Waiting in Concurrent Algorithms. In *4th international conference on networked systems (NETYS 2016)*, Marrakech, Morocco, May 2016.

[39] J. D. Valois. Implementing lock-free queues. In *Proc. of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 212–222, 1994.