Fair Synchronization

Gadi Taubenfeld

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

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. 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 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 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.

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 [17] and non-blocking [19]. 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. 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, critical 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: a loop that includes 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 critical section at some point in time, if sufficiently many steps of that process will carry it into the critical 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 to enter its critical 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 critical section, if a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical 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 critical section twice before a waiting process completes executing its critical 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 critical 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 critical 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 critical sections at the same time as long as fairness is preserved. We notice that no lock implementation may satisfy the concurrency requirement.

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 critical sections. The waiting processes which are not enabled will eventually simultaneously become enabled, before or once the currently enabled processes exit their critical and exit sections. We observe that the stronger FIFO fairness requirement, the progress requirement and concurrency requirement cannot be mutually satisfied (see [36] for a proof).

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 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 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:

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 explained 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 algorithm, in addition to satisfying the mutual exclusion and deadlock freedom requirements (Section 4), 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 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.

Related Work

Mutual exclusion locks were first introduced by Edsger W. Dijkstra in [8]. Since than, numerous implementations of locks have been proposed [30, 32]. Various other types of locks, like ℓ-exclusion locks [13, 12] 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 2, uses some ideas from the mutual exclusion algorithm presented in [33].

Implementations of data structures which avoid locking have appeared in many papers, a few examples are [9, 14, 15, 27, 31, 37]. Several progress conditions have been proposed for data structures which avoid locking. The most extensively studied conditions are wait-freedom [17] and non-blocking [19]. Strategies that avoid locks are called lockless [16] or lock-free [26]. (In some papers, lock-free means non-blocking.) Consistency conditions for concurrent objects are linearizability [19] and sequential consistency [22]. A tutorial on memory consistency models can be found in [1].

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) [18]. Transformations that automatically convert any obstruction-free algorithm into a non-blocking or a wait-free algorithm are presented in [10, 34], 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 the common cases, when there is no contention, or when processes with non-interfering operations access it concurrently, are introduced in [35]. 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 [29].

2 The Fair Synchronization Algorithm

We use one (multi-writer multi-reader) atomic bit, called *queue*. The first thing that process i does in its entry section is to read the value of the *queue* bit, and to determine to which of the two queues (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 queue, it waits until its queue has priority over the other queue and then it enters its critical section. The order in which processes can enter their critical sections is defined as follows: If two processes belong to different queues, the process whose queue, as recorded in its *state* register, is *different* from the value of the bit *queue* is enabled and can enter its critical section, and the other process has to wait. If all the active processes belong to the same queue then they can all enter their critical sections.

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

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

The following registers are used: (1) a single multi-writer atomic bit named *queue*, (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 *queue*) tries to decide to which of the two queues, 0 or 1, it should belong. When $state_i = 1$, process i is active and belongs to queue 1. When $state_i = 0$, process i is active and belongs to queue 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. Finally, whenever two atomic registers appear in the same statement, two separate steps are required to execute this statement. The algorithm is given below.[1]

Algorithm 1. A FAIR SYNCHRONIZATION ALGORITHM: `process` i`'s code` $(1 \leq i \leq n)$

Shared variables:

$queue$: atomic bit; the initial value of the queue 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 $state_i := queue$ `/* choose queue and end doorway */`
3 **for** $j = 1$ **to** n **do** `/* begin waiting */`
4   **if** $(state_i \neq queue)$ **then** break **fi** `/* process is enabled */`
5   **await** $state_j \neq 2$
6   **if** $state_j = 1 - state_i$ `/* different queues */`
7   **then await** $(state_j \neq 1 - state_i) \vee (state_i \neq queue)$ **fi**
8 **od** `/* end waiting */`
9 *critical section*
10 $queue := 1 - state_i$ `/* 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 *queue* 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 queue to which it belongs are simultaneously enabled and then it enters its critical section. This happens when either, $(state_i \neq queue)$, i.e. the value the *queue* bit points to the queue which i does *not* belong to (line 4), or when all the waiting processes (including i) belong to the same queue (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 queues (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 *queue* bit.

In the exit code, i sets the *queue* bit to a value which is different than the queue 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 queue bit immediately before it enters its critical section. The order of lines 10 and 11 is crucial for correctness.

We observe that a *non* beginning process, say p, may enter its critical section ahead of another waiting process, say q, twice: the first time if p is enabled on the other queue, and the second time if p just happened to pass q which is waiting on the same queue and enters its critical section first. We point out that omitting lines 1 and 5 will result

[1] 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.

an incorrect solution. It is possible to replace each one of the 4-valued single-writer atomic registers, by three *separate* atomic bits. In the full version [36], 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 which its time complexity, in the absence of contention, is a constant [24]. 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. 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 processes can safely enter its critical 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 which its 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 [21] into its corresponding adaptive version using the active set object. In [33], 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 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 queue a process belongs to, on the order in which processes enter their critical sections.

Lemma 1. *For any two waiting processes i and j, if $state_i \neq queue$ and $state_j = queue$, then i must enter its critical section and complete its exit section before j can enter its critical section.*

Proof. A waiting process, say i, is *enabled* to enter its critical section only when one of the following two condition holds: (1) the value of $state_i \neq queue$. 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 queue than the queue i belongs to. In such a case, i will execute the loop n times and will exit. If non of these

two conditions holds, i will eventually have to wait in line 7, until either the value of the queue bit changes of the processes which belong to the other queue change the values of their state registers.

Until the value of the *queue* bit is first changed, all the active processes belong to the same queue $v \in \{0, 1\}$. Hence, as explained above, they are all enabled. The first process to finish its critical section flips the value of the *queue* bit and sets it $1 - v$. Thereafter, the value of the queue bit read by all the new beginning processes is $1-v$. As explained in the previous paragraph, non of these new beginning processes can become enabled until either, the value of the queue bit changes again, or all the processes which belong to the queue v complete their exit sections. Since all the processes which belong to the queue v set the queue 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 queue $1 - v$, and hence will all become enabled. When they exit they change back to v the value of the *queue* bit, and so on. As we can see in the above explanation, for any two waiting processes i and j, if $state_i \neq queue$ and $state_j = queue$, then i is enabled and j is disabled, and i and all the processes which belong to the same queue as i will enter their critical sections and complete their critical and exit sections before j can enter its critical section. □

Proof of Theorem 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 critical section. It follows from Lemma 1, that this can happen only if both i and j belong to the same queue (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 *queue* bit to $1 - state_i$. Thereafter, by Lemma 1, the value of the queue bit will not change (at least) until j completes its exit section. If i will try to enter its critical section again while j has not completed its exit section yet, then after passing through its doorway i will belong to a different queue than j (i.e., $state_i \neq state_j$) and the value of the *queue* bit will be the same as the value of $state_i$. Thus, by Lemma 1, i will not again become enabled 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 to the contrary that the algorithm does not satisfy progress and show how this assumption leads to a contradiction. Assuming that the algorithm does not satisfy progress means that 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 critical sections. In the later case, all the processes which their state register is different from the value of the queue bit can proceed to their critical 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 1, a waiting process, say i, is *enabled* to enter its critical section only when one of the following two condition holds: (1) the value of $state_i \neq queue$; 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 queue than the queue 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 queue bit has changed then also j must become enabled for that reason. If i becomes enabled because no process belongs to a different queue than the queue i belongs to, then also j must become enabled for that reason. Thus, the algorithm satisfies *concurrency*. □

3 Fair Data Structures

In order to impose fairness on a concurrent date structure, concurrent accesses to a data structure can be synchronized using a fair synchronization algorithm: a process accesses the data structure only inside a critical section code. 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*.[2] 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 one for another, precluding disjoint-access parallelism. This can be resolved when using fine-grained fair synchronization.

3.1 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.

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

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.*

To keep things 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.

3.2 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 [19]. 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 2. *Let A be a fair synchronization algorithm and let B be a non-blocking or a wait-free data structure. Assume that the registers of A are different from the registers of B. Let C be a data structure obtained by replacing the critical section of A with the data structure B. Then, C is a fair data structure. Furthermore, if B is linearizable, then also C is linearizable.*

The correctness proof appears in [36]. Using Theorem 2, it is now possible to construct new fair data structures from existing non-blocking or wait-free data structures.

4 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]. As before, 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.

4.1 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 terms 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.

4.2 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 3. *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 critical 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.*

The correctness proof appears in [36]. Using Theorem 3, 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 [23], is a deadlock-free mutual exclusion algorithm for n processes which uses n shared bits. By Theorem 3, using the fair synchronization algorithm from Section 2 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 a $3n+1$ bits.

Several techniques for designing FIFO mutual exclusion algorithms have been used in [20, 23, 25]. 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 *queue* once), the doorway of the various FIFO mutual exclusion algorithms [20, 23, 25] is not fast wait-free as it takes at least n steps, where n is the number of processes. Next we use Theorem 3 for proving a space lower bound for the fair synchronization problem.

5 A Space Lower Bound for Fair Synchronization

In Section 2, 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*. The value *val* is assigned to r, and the old 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 4. *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 3, 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 satisfy 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 [28], 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 [11] 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, 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. □

6 Discussion

We have considered the problem of enforcing fairness in a shared-memory algorithm, by preventing a process from accessing a shared resource twice while another process is waiting to get the resource. We have proposed to enforce fairness as a wrapper around any concurrent algorithm, and studied the consequences. We have formalized the fair synchronization problem, presented a solution, and then showed that existing concurrent data structures and mutual exclusion algorithms can be encapsulated into a fair synchronization construct to yield algorithms that are inherently fair. A linear space lower bound has been obtained for the problem.

Wait-free algorithms are frequently criticized for sacrificing performance compared to non-blocking algorithms. When enforce fairness as a wrapper around a concurrent algorithm, it is better that the concurrent algorithm be an efficient non-blocking algorithms rather than a wait-free algorithm. Since many processes may enter their critical 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 critical 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: If strictly fewer than ℓ processes fail (are delayed forever) then if a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical 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. It seems that allowing limited amounts of overtaking (e.g., a process accessing a shared resource for a constant number of times while another is spinning on it) would not be detrimental. 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 4) 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.

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. W. B. Easton. Process synchronization without long-term interlock. In *Proc. of the 3rd ACM symp. on Operating systems principles*, pages 95–100, 1971.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *15th international symposium on distributed computing*, October 2001.
16. 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.

17. M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.
18. 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.
19. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *toplas*, 12(3):463–492, 1990.
20. 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.
21. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
22. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 28(9):690–691, September 1979.
23. L. Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the ACM*, 33:327–348, 1986.
24. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.
25. 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.
26. H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.
27. 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.
28. G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).
29. M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer. ISBN 978-3-642-32027-9, 515 pages, 2013.
30. M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.
31. 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.
32. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall. ISBN 0-131-97259-6, 423 pages, 2006.
33. G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 56–70.
34. 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.
35. 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.
36. G. Taubenfeld. Fair Synchronization, 2013. The full version is avaliable at: http://www.faculty.idc.ac.il/gadi/Publications.htm.
37. J. D. Valois. Implementing lock-free queues. In *Proc. of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 212–222, 1994.