J. Parallel Distrib. Comput. 97 (2016) 1–10

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Fair synchronization☆

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel

HIGHLIGHTS

- We define a new important synchronization problem – called fair synchronization – for concurrent programming.
- We present the first fair synchronization algorithm for n processes.
- We define the notion of a fair data structure and show how to implement such data structures.
- We prove that by composing a fair synchronization algorithm and an unfair lock, it is possible to construct a fair lock.
- We show that $n - 1$ registers and conditional objects are necessary for solving the fair synchronization problem for n processes.

ARTICLE INFO

Article history:
Received 27 August 2015
Received in revised form
28 May 2016
Accepted 16 June 2016
Available online 24 June 2016

Keywords:
Synchronization
Fairness
Concurrent data structures
Non-blocking
Wait-freedom
Locks
Mutual exclusion

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.

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

☆ 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 Taubenfeld (2009).

E-mail address: tgadi@idc.ac.il.

http://dx.doi.org/10.1016/j.jpdc.2016.06.007
0743-7315/

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.