Looking for Efficient Implementations of Concurrent Objects

Achour Mostefaoui, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1969 – Mars 2011

Looking for Efficient Implementations of Concurrent Objects

Achour Mostefaoui[*] Michel Raynal[**]

Abstract: As introduced by Taubenfeld, a contention-sensitive implementation of a concurrent object is an implementation such that the overhead introduced by locking is eliminated in the common cases, i.e., when there is no contention or when the operations accessing concurrently the object are non-interfering. This paper, that can be considered as an introductory paper to this topic, presents a methodological construction of a contention-sensitive implementation of a concurrent stack. In a contention-free context a push or pop operation does not rest on a lock mechanism and needs only six accesses to the shared memory. In case of concurrency a single lock is required. Moreover, the implementation is starvation-free (any operation is eventually executed). The paper, that presents the algorithms in an incremental way, visits also a family of liveness conditions and important concurrency-related concepts such as the notion of an abortable object.

Key-words: Abortable object, Asynchronous shared memory system, Atomic register, Compare&Swap, Contention manager, Contention-sensitiveness, Deadlock-freedom, Linearizability, Liveness, Lock-freedom, Non-blocking, Obstruction-freedom, Progress condition, Starvation-freedom, Synchronization.

Sur la qualité de service des objects concurrents

Résumé : *Au delà de l'aspect recherche de solution de faisabilité, ce rapport surtout à la recherche de mise en œuvres efficaces d'objets concurrents.*

Mots clés : *Système asynchrone, mémoire partagée, objet concurrent, tolérance aux fautes.*

[*] Membre du Projet ASAP: équipe commune avec l'INRIA, le CNRS et l'université Rennes 1, `achour@irisa.fr`.
[**] Membre senior de l'IUF et Projet ASAP: équipe commune avec l'INRIA, le CNRS et l'université Rennes 1, `raynal@irisa.fr`.

1 Introduction

1.1 Concurrent objects

From mastering sequential algorithms to mastering concurrency The study of algorithms lies at the core of informatics and participates in establishing it as a *science* with strong results on what can be computed (decidability) and what can be efficiently computed (complexity). It is consequently unanimously accepted by the community that any curriculum for undergraduate students has to include lectures on sequential algorithms. This allows the students not only to better master the basic concepts, mechanisms, techniques, difficulties and subtleties that underlie the design of algorithms, but also understand the deep nature of computer science and computer engineering.

A challenge is now to attain the same goal in the context of concurrency. A *concurrent object* is an object that can be concurrently accessed by several processes. As any object, a concurrent object is defined by a set of operations that processes can invoke to cooperate through this object. These operations are the only way to access the internal representation of the object (that remains otherwise invisible to processes). We are interested here in concurrent objects that have a *sequential specification* and supply processes with *total operations*. A total operation is an operation that always returns a result (e.g., instead of blocking the invoking process, a dequeue() operation on an empty queue returns it the value $empty$).

Linearizability The most popular safety property associated with concurrent objects is called *linearizability* [10]. This consistency condition extends *atomicity* to all objects defined by a sequential specification on total operations. More precisely, an implementation of an object satisfies *linearizability* (and we say that the object implementation is *linearizable*) if the operation invocations issued by the processes appear (from an external observer point of view) as if they have been executed sequentially, each invocation appearing as being executed instantaneously at some point of the time line between its start event and its end event. Said differently, an implementation is linearizable if it could have been produced by a sequential execution.

An important property associated with linearizable object implementations is that they compose for free. This means that, if both of the implementation of an object A and the implementation of an object B (each taken independently) are linearizable, then these implementations without any modification constitute a linearizable implementation of the composite object (A, B). (It is important to notice that, in contrast to linearizability, other consistency conditions such as sequential consistency [14] or serializability [2] cannot be composed for free.)

Traditional lock-based shared memory synchronization One of the most popular way to obtain linearizable implementations of concurrent objects is to use locks. Associating a single lock with an object prevents several processes/threads from accessing it simultaneously. This approach is based on the classical notion of mutual exclusion [3, 18, 24]. Interestingly, locks can take different shapes according to the abstraction level at which they are considered. The most known example of locks is certainly the *semaphore* object [3], on top of which more friendly (i.e., high level) lock-based abstractions (such as monitors [12] or serializers [11]) can be built. This approach has proved its usefulness in providing simple lock-based solutions to basic paradigms of shared memory synchronization (such as the producer-consumer problem, or the readers-writers problem). One of the main difficulties when designing a lock-based solution lies in ensuring deadlock prevention, and more generally, provable liveness guarantees. Moreover, from an implementation point of view, lock implementations can be costly in terms of underlying shared memory accesses [19].

Contention-sensitive objects The notion of *contention-sensitive implementation* of a concurrent object has been recently introduced [26]. The contention-sensitiveness property means that the overhead due to locking has to be eliminated when there is no concurrency or when the operations that concurrently access an object are not interfering (e.g., enqueing and dequeuing on a non-empty queue). In these cases (absence of contention or interference), a contention-sensitive implementation has to ensure that an operation on the object completes in a small (possibly constant) number of steps and without locks. Resorting to locks has to be restricted to concurrent conflicting operations only.

The first paper (to our knowledge) that introduced contention-sensitiveness (without giving it a name) is [16] where is presented a mutual exclusion algorithm in which, in a contention-free context, a process has to execute only seven shared memory accesses to enter the critical section. When there is contention, the number of shared memory accesses depends on the number of processes and the actual concurrency pattern.

1.2 Content of the paper

Abortable objects An *abortable* concurrent object behaves like an ordinary object when accessed sequentially, but may abort operations when accessed concurrently (in that case the aborted operation has no effect and returns a default value denoted $\perp$). This definition is inspired from, but stronger than, the definition of abortable objects introduced in [1] (in that paper, an aborted operation returns also $\perp$, but may or not take effect and this is not known by the invoking process). The important point (in both definitions) is that the state of the object is never left inconsistent.

As far as we know, the notion of abortable objects has first been discussed in [13] where is presented an abortable mutual exclusion object. At any time while it is executing its entry code, a process can stop competing for the critical section and this halting has not to alter the liveness of the other critical section requests.

Progress conditions While it always considers linearizability as the implicit safety condition, this paper considers three progress conditions for concurrent objects: obstruction-freedom, non-blocking and starvation-freedom.

The *obstruction-freedom* progress condition [8] states that an operation is required to terminate only if it executes in a concurrency-free context (i.e., when there is no operation invoked concurrently which is also called *solo* execution). Hence, an obstruction-free implementation of an object does not prevent concurrent operation invocations from never terminating. Let us notice that the notion of an abortable object is stronger than obstruction-freedom: while both ensure object consistency, they differ in the liveness they provide to users. More precisely, both guarantee operation termination in concurrency-free context, obstruction-freedom does not guarantee operation termination in case of concurrency. Differently, all operation invocations of an abortable object do terminate (possibly returning the value $\perp$ in case of concurrency). Hence, an implementation of an abortable object trivially satisfies the obstruction-freedom progress condition while the opposite is not true.

An implementation of a concurrent object is *non-blocking* if it is obstruction-free and additionally guarantees that, in presence of concurrency, at least one concurrent operation terminates. In a failure-free context, non-blocking is the same as *deadlock-freedom*. Finally, an implementation of a concurrent object is *starvation-free* if any operation invoked by a process terminates[1]. Hence, we have a hierarchy of progress conditions: obstruction-freedom is strictly weaker than non-blocking that in turn is strictly weaker than starvation-freedom. This hierarchy defines a family of qualities of service for liveness properties.

Content and roadmap This paper investigates the contention-sensitive approach for the implementation of concurrent objects as advocated by Taubenfeld in [26]. To that end, it considers a simple concurrent object, namely a shared task (let us remark that a lock-based starvation-free implementation of such an object is trivial). Three algorithms implementing such an object are presented. The first algorithm provides the processes with an abortable stack. As already said, this means that concurrent push and pop operations are allowed to abort (i.e., return $\perp$), while a push or pop operation executed in a concurrency-free context has to terminate and return a non-$\perp$ value. This algorithm does not use locks and is consequently *lock-free*. The second algorithm, which is also lock-free and provides the processes with a non-blocking shared stack is a simple extension of the previous one.

Considering an underlying abortable shared stack, the third algorithm provides the processes with a contention-sensitive shared stack. When an operation is executed in a concurrency-free context, this algorithm uses no lock and, whatever the number of processes and the size of the stack, it requires only seven shared memory accesses. This means that the algorithm is particularly efficient in contention-free patterns. It resorts to a lock only when there are concurrent operations. Moreover, this algorithm ensures the starvation-freedom progress condition.

The algorithms are built incrementally. This helps better understand the mechanisms that are used to go from an abortable shared object to a contention-sensitive implementation that satisfies the starvation-freedom progress condition. Interestingly, the mechanism employed to ensure starvation-freedom constitute a *contention manager* that can be used to solve other fairness-related problems.

The paper is made up of 5 sections. Section 2 presents the computation model. Then Section 3 presents an algorithm implementing an abortable stack object and its extension to obtain a non-blocking implementation of a stack [22]. Section 4 presents a contention-sensitive algorithm that implements a starvation-free stack. This algorithm is based on a mechanism introduced in [26]. Finally, Section 5 concludes the paper. Last but not least, it is important to say that the aim of this paper is to promote the notion of contention-sensitive implementation of a concurrent object as an efficient alternative to fully lock-based implementations. The interested reader will find more general developments on the contention-sensitive approach in [26].

2 Computation model

2.1 System model

Asynchronous processes and communication model The system is made up of n sequential processes denoted $p_1, p_2, \ldots, p_n$. The integer i is the identity of p_i. Each process proceeds to its own speed, which means that the processes are asynchronous.

Processes communicate by accessing a shared memory that consists of *atomic* registers. The base operations on a register are read, write and Compare&Swap (see below). "Atomic" means that all operations on a register R appear as if they have been executed sequentially, and if operation $op1$ terminates before operation $op2$ starts, then $op1$ appears before $op2$ in the sequence.

Atomicity and linearizability denote the same consistency condition. The word "atomicity" is usually employed for read/write registers [15] while the word "linearizability" is employed for objects built on top of registers or other objects [10].

[1] In presence of process crashes, starvation-freedom becomes *t-resilience* where t is the maximum number of process that may crash. Moreover, in a set of n processes, *wait-freedom* [7] is $(n-1)$-resilience.

Notation Shared registers are denoted with uppercase letters. In contrast, variables that are local to a process are denoted with lowercase letters.

Failure model It is assumed that both processes and atomic registers are reliable. This helps better understand how the algorithms work. They actually can cope with process crash failures. This is shortly discussed in Section 5.

2.2 Compare&Swap operation

Definition The Compare&Swap operation, that is on an atomic register X is denoted $X.\mathsf{C\&S}(old, new)$. It is a conditional write that does atomically the following: if the current value of X is old, it assigns new to X and returns $true$; otherwise, it returns $false$.

```
primitive $X.\mathsf{C\&S}(old, new)$:
    if $(X = old)$ then $X \leftarrow new$; return($true$) else return($false$) end if.
```

This base operation exists on some machines such as Motorola 680x0, Intel, Sun, IBM 370 and SPARC architectures. In some cases, the returned value is not a boolean, but the previous value of X.

The ABA problem When using Compare&Swap, a process p_i usually does the following. It first reads the atomic register X (obtaining value a) and later wants to update X to a new value c only if X has not been modified by another process since it has been read by p_i. Hence, p_i invokes $X.\mathsf{C\&S}(a, c)$. Unfortunately, the fact that this invocation returns $true$ to p_i does not allow it to conclude that X has not been modified since the last time it read it. This is because between the read of X and the invocation $X.\mathsf{C\&S}(a, c)$ issued by p_i, X may have been updated twice, first by a process p_j that has successfully invoked $X.\mathsf{C\&S}(a, b)$ and then by a process p_k that has successfully invoked $X.\mathsf{C\&S}(b, a)$, thereby restoring the value a into X. This is called the ABA problem.

This problem can be solved by associating a new (tag) sequence number with each value that is written. The atomic register X is then composed of several fields such as $\langle v, sn\rangle$ where v is the current value of X and sn its associated sequence number. When it reads X a process p_i obtains consequently the pair $\langle v, sn\rangle$. When later it wants to conditionally writes v' into X, it invokes $X.\mathsf{C\&S}(\langle v, sn\rangle, \langle v', sn + 1\rangle)$. It is easy to see that the write succeeds only if X has continuously been equal to $\langle v, sn\rangle$.

3 Implementing an abortable stack and a non-blocking stack

The algorithm described in Figure 1 implements an abortable stack. It is a simplified version of the non-blocking algorithm introduced in [22] (which is presented in Figure 2).

Operations An abortable stack has two operations denoted here $\mathsf{weak\_push}(v)$ (where v is the value to be added at the top of the stack) and $\mathsf{weak\_pop}()$. An operation always succeeds when executed in a contention-free context. In that case $\mathsf{weak\_push}(v)$ returns $done$ if v has been pushed on the stack and *full* if the stack is full; $\mathsf{weak\_pop}()$ returns the value that was at the top of the stack (and suppresses it from the stack) or returns $empty$ if the stack is empty. In the other cases, an operation may abort, in which case it returns $\perp$.

Shared data structures The stack is implemented with an atomic register denoted TOP and an array of $k+1$ atomic registers denoted $STACK[0..k]$.

- TOP has three fields that contain an index (to address an entry of $STACK$), a value and a counter. It is initialized to $\langle 0, \perp, 0\rangle$.
- Each atomic register $STACK[x]$ has two fields: $STACK[x].val$ that contains a value, and $STACK[x].sn$ that contains a sequence number (used to prevent the ABA problem as far as $STACK[x]$ is concerned).

  The capacity of the stack is k and for $1 \leq x \leq k$ the register $STACK[x]$ is initialized to $\langle \perp, 0\rangle$. $STACK[0]$ is a dummy entry initialized to $\langle \perp, -1\rangle$ that always contains the default value $\perp$.

The array $STACK$ is used to store the content of the stack, and the register TOP is used to store the index and the value of the element at the top of the stack. The content of both TOP and $STACK[x]$ is modified with the help of the Compare&Swap operation. This operation is used to prevent erroneous modifications of the stack internal presentation.

The implementation is *lazy* in the sense that a stack operation assigns its new value to TOP and leave the corresponding modification of $STACK$ to the next stack operation. Hence, while on the one hand a stack operation is lazy, on the other hand it has to help terminate the previous stack operation.

```
operation weak_push($v$):
(01) $(index, value, seqnb) \leftarrow TOP$;
(02) help$(index, value, seqnb)$;
(03) if $(index = k)$ then return$(full)$ end if;
(04) $sn\_of\_next \leftarrow STACK[index+1].sn$;
(05) $newtop \leftarrow \langle index+1, v, sn\_of\_next+1\rangle$;
(06) if $TOP$.C&S$(\langle index, value, seqnb\rangle, newtop)$
(07)         then return$(done)$ else return$(\perp)$ end if.

operation weak_pop():
(08) $(index, value, seqnb) \leftarrow TOP$;
(09) help$(index, value, seqnb)$;
(10) if $(index = 0)$ then return$(empty)$ end if;
(11) $belowtop \leftarrow STACK[index-1]$;
(12) $newtop \leftarrow \langle index-1, belowtop.val, belowtop.sn+1\rangle$;
(13) if $TOP$.C&S$(\langle index, value, seqnb\rangle, newtop)$
(14)         then return$(value)$ else return$(\perp)$ end if.

procedure help$(index, value, seqnb)$:
(15) $stacktop \leftarrow STACK[index].val$;
(16) $STACK[index]$.C&S$(\langle stacktop, seqnb-1\rangle, \langle value, seqnb\rangle)$.
```

Figure 1: An abortable stack [22]

The operation weak_push(v) When a process p_i invokes weak_push(v), it first reads the content of TOP (that contains the last non-aborted operation on the stack) and stores its three fields in its local variables $index$, $value$ and $seqnb$ (line 01).

Then, p_i helps terminate the previous non-aborted stack operation (line 02). That operation (be it a successful weak_push() or a successful weak_pop() as we will see later) required to write $\langle value, seqnb\rangle$ into $STACK[index]$. To that end p_i invokes $STACK[index]$.C&S.(old, new) with the appropriate values old and new in order the write be executed only if not yet done (lines 15-16).

After its help (that was successful if not yet done by another stack operation) to move the content of TOP into $STACK[index]$, p_i returns *full* if the stack is full (line 03). If the stack is not full, it tries to modify TOP to register its push operation. This operation has to succeed if no other process modified TOP since it was read by p_i at line 01. In that case, TOP takes its new value and weak_push(v) succeeds. Otherwise it aborts (lines 06-07).

The triple of values associated with this push_try(v) and to be written in TOP if successful, is computed at lines (lines 04-05). Process p_i first computes the last sequence number sn_of_next used in $STACK[index+1]$ and then defines the new triple, namely, $newtop = \langle index+1, v, sn\_of\_next+1\rangle$ to be written first in TOP and later in $STACK[index+1]$ thanks to the help provided by the next stack operation (let us remember that sn_of_next+1 is used to prevent the ABA problem).

The operation weak_pop() The algorithm implementing this operation has exactly the same structure as the previous one and is nearly the same. Its explanation is consequently left to the reader.

Linearization points of successful weak_push() **and** weak_pop() **operations** The operations that do not abort are linearizable, i.e., they can be totally ordered on the time line, each operation being associated with a single point of the time line that is after its start event and before its end event. More precisely, a non-aborted operation appears as if it has been atomically executed

- when it reads TOP (at line 01 or 08) if it returns *full* or *empty* (at line 03 or 10),
- or at the time at which it successfully executes TOP.C&S$(-,-)$ (line 06 or 13 according to the operation).

From an abortable stack to a non-blocking stack A very simple algorithm that builds a non-blocking stack on top of an abortable stack is described in Figure 2. It is easy to see that this algorithm satisfies the obstruction-freedom property: an operation executed in a contention-free context returns always a non-$\perp$ value. It is also easy to see that no operation aborts: instead of aborting, an operation can loop forever. The interested reader will find in [22] a proof that, whatever the contention pattern, at least one operation always terminates (i.e., the algorithm is non-blocking).

```
operation non_blocking_push($v$):
    repeat $res \leftarrow$ weak_push($v$) until $res \neq \perp$ end repeat;
    return($res$).

operation non_blocking_pop():
    repeat $res \leftarrow$ weak_pop() until $res \neq \perp$ end repeat;
    return($res$).
```

Figure 2: A linearizable non-blocking stack

4 A contention-sensitive implementation of stack

Let us remember that the aim is here the design of a contention-sensitive that implements a starvation-free stack, which means that the algorithm (a) is allowed to use a lock only when there is contention, and (b) has to execute a small and constant-bounded number of shared memory accesses when there is no contention.

The stack provides the processes with the operations denoted strong_push(v) and strong_pop(). As the implementation of the contention-sensitiveness property is independent of the fact that the stack operation is strong_push() or strong_pop(), we describe a generic algorithm denoted strong_push_or_pop(par) where $par = v$ if the operation is strong_push(v) and $par = \perp$ if the operation is strong_pop(). Moreover, in the text of the algorithm weak_push_or_pop(par) stands for weak_push(v) or weak_pop() according to the context.

4.1 Data structures

The implementation of the contention-sensitiveness property is based on two atomic registers, an array of atomic registers and a lock.

- The lock, denoted $LOCK$, is accessed by the operations lock() and unlock(). It is used to ensure that a single process executes the part of code bracketed by $LOCK$.lock() and $LOCK$.unlock(). This lock is assumed to be deadlock-free but it is not required to be starvation-free (see the remark below).
- $CONTENTION$ is a boolean register (initialized to *false*) that is set to *true* by a process when it executes the underlying weak_operation(par) operation. This allows a process that starts executing an operation to know that there is contention.
- $FLAG[i]$ boolean, is a boolean (initialized to *false*) that process p_i sets to *true* when it wants to execute a stack operation and there is contention. In that way, p_i allows the other processes to know it is competing for the lock. Process p_i sets $FLAG[i]$ to *false* when it has executed its base weak_operation(par) operation.
- $TURN$ contains a process identity. $TURN = i$ means that process p_i has priority to use the lock. Its initial value is any process identity. In order to ensure starvation freedom, the next value of $TURN$ is $(TURN \mod n) + 1$. Such a round-robin mechanism is used in several mutual exclusion algorithms such as [17, 23].

Remark If the lock is starvation-free (i.e., it ensures that any requesting process will obtain the lock) the algorithm can be simplified. More precisely, the array $FLAG[1..n]$ and the register $TURN$ become useless and consequently the lines 04-05 and 10-11 can be suppressed from algorithm. Those are actually shared variables and the associated statements that transform a deadlock-free lock into a starvation-free lock.

4.2 The algorithm

The algorithm is described in Figure 3. It is made of two parts. A lock-free part and a lock-based part. (The lock-free part is called *shortcut* in [26].)

In the first part (lines 01-03), the invoking process p_i reads $CONTENTION$ and, if this boolean is false, invokes the underlying weak_operation() operation. As we have seen if there is no contention this invocation returns a non-$\perp$ value and p_i terminates. The number of shared memory accesses is then 6 (5 within the successful weak_push_or_pop() + 1 for the read of $CONTENTION$).

If $CONTENTION$ is equal to *true* or weak_push_or_pop() returns $\perp$, p_i knows there is contention. In that case, p_i enters the second part (lines 04-13) which is made up of two phases.

- In the first phase (lines 04-05), p_i first sets $FLAG[i]$ to *true* to inform the other processes that it is competing for the critical section protected by the lock. Then, p_i waits until either it is the process that is currently given priority ($TURN = i$) or the process that is currently given priority (namely p_{TURN}) is not competing ($FLAG[TURN] = false$). When one of these predicates becomes true, p_i invokes $LOCK$.lock().

```
operation strong_push_or_pop($par$):   % $par = v$ for push() and $\perp$ for pop() %
(01)   if ($\neg CONTENTION$)
(02)      then $res \leftarrow$ weak_push_or_pop($par$); if ($res \neq \perp$) then return($res$) end if
(03)   end if;
(04)*  $FLAG[i] \leftarrow true$;
(05)*  wait $((TURN = i) \vee (\neg FLAG[TURN]))$;
(06)*  $LOCK$.lock();
(07)   $CONTENTION \leftarrow true$;
(08)   repeat $res \leftarrow$ weak_push_or_pop($par$) until $res \neq \perp$ end repeat;
(09)   $CONTENTION \leftarrow false$;
(10)*  $FLAG[i] \leftarrow false$;
(11)*  if ($\neg FLAG[TURN]$) then $TURN \leftarrow (TURN \mod n) + 1$ end if;
(12)*  $LOCK$.unlock();
(13)   return($res$).
```

Figure 3: A linearizable contention-sensitive starvation-free stack (code for p_i)

- The second phase (lines 06-13) starts when p_i has gained mutual exclusion and is consequently the only process executing the lines 07-12.

  Process p_i executes then repeatedly weak_push_or_pop(par) until a successful invocation (line 08). When, this occurs it resets $CONTENTION$ and $FLAG[i]$ to *false*. Then if the process p_{TURN} that is currently given priority is not competing ($FLAG[TURN] = false$), p_i gives priority to $p_{(TURN \mod n)+1}$ (line 11) before releasing the lock and returning its (non-$\perp$) result.

  It is important to notice that, due to asynchrony and the code of lines 01-03, while a process p_i is repeatedly executing weak_push_or_pop() at line 08, other processes can be executing weak_push_or_pop() at line 02 (because they read *false* from $CONTENTION$) and the execution of weak_push_or_pop() by these processes can be successful. As we will see in the proof, this does not cause a problem because (a) the number of strong_push_or_pop() invocations concurrent with the one of p_i is bounded and (b) the future invocations of strong_push_or_pop() will read *true* from $CONTENTION$) and will consequently enter the second part of the algorithm in which they cannot bypass p_i.

4.3 Proof

Lemma 1 *If a process p_i returns from its* strong_push(v) *or* strong_pop() *invocation, it returns a non-$\perp$ value.*

Proof The proof follows immediately from the predicate $res \neq \perp$ tested at line 02 if p_i returns at that line, or tested at line 08 if p_i returns line 13. $\Box_{Lemma\ 1}$

Lemma 2 *If a process p_i eventually succeeds in locking, it eventually terminates its current* strong_push() *or* strong_pop() *operation.*

Proof Let us assume that a process p_i succeeds in locking at time t_1. There is a consequently a finite time $t_2 > t_1$ from which $CONTENTION$ is $true$.

It follows that all the processes that invoke strong_push(v) or strong_pop() after time t_2 skip the lock-free part and start competing for the lock after it has been acquired by p_i. Hence these processes cannot prevent p_i from terminating its operation.

It follows that at most x processes, $0 \leq x \leq n-1$, can be executing weak_push() or weak_pop() at line 02 while p_i is executing weak_push() or weak_pop() at line 08. Let X the corresponding set of processes. If $X = \emptyset$, the execution by p_i of weak_push() or weak_pop() is concurrency-free and the the lemma trivially follows. Hence, let us consider the case $X \neq \emptyset$, As we have seen in Section 3, the processes in X eventually terminate their executions of weak_push() or weak_pop(). In the worst case, p_i loops executing weak_push() or weak_pop() (at line 08) until all the processes in X have returned from their current invocation of weak_push() or weak_pop() at line 02. Let t_3 be such a time instant. If p_i has not returned from its weak_push() or weak_pop() operation with a non-$\perp$ value before t_3, it follows from the previous observation that its first invocation of weak_push() or weak_pop() after t_3 will return a non-$\perp$ value, and consequently p_i eventually terminates. $\Box_{Lemma\ 2}$

Lemma 3 *If, while executing a* strong_push(v) *or* strong_pop() *operation, a process p_i reads $true$ from $CONTENTION$ at line 01 or obtains $res = \perp$ at line 02, it eventually obtains the lock.*

Proof Let us consider a process p_i that sets $FLAG[i]$ to $true$ (line 04). Hence, p_i is a process as defined by the lemma assumption. We consider three cases.

1. Process p_i exits the loop of line 05 because $TURN = i$.

   Let us observe that, in this case, $TURN$ remains equal to i until p_i resets $FLAG[i]$ to *false* (line 10) and increases $TURN$ to $(i \mod n) + 1$ (line 11).

   It follows from the previous observation that any process p_j $(j \neq i)$ that executes the loop of line 05 after $TURN$ has been set to i, loops until p_i executes the lines 10-11. Let Y be this (possibly empty) set of processes.

   Hence, at most x processes, $0 \leq x \leq n - (|Y| + 1)$, can compete with p_i for obtaining the lock. As the lock is deadlock-free, it follows that, in the worst case, each of these processes obtains the lock before p_i. After they have obtained (and released) the lock, p_i is the only process requesting the lock and necessarily obtains it, which completes the proof of the lemma for this case.

2. Process p_i exits the loop of line 05 because $TURN = k \neq i$ and $FLAG[k]$ is equal to *false*. We have to show that p_i eventually obtains the lock.

   Let us assume by contradiction that p_i never obtains the lock. In the worst case, all processes are competing with p_i to obtain the lock. Let p_j be the process that obtains the lock (as the lock is deadlock-free, such a process does exist). Due to Lemma 2, it follows that p_j eventually releases the lock. If $FLAG[TURN] =$ *false*, p_j advances $TURN$ to its successor p_ℓ (line 11) along the oriented logical ring $j, j + 1, \ldots, n, 1, \ldots$. We have then $TURN = \ell$. If $\ell \neq i$, the reasoning is repeated replacing p_j by p_ℓ (let us observe that, due a reasoning similar to Item 1, p_ℓ eventually obtains the look). As no process is skipped when $TURN$ is advanced to its successor, it follows that $TURN$ progresses from process to process until w have $TURN = i$. When this occurs, all processes that execute line 05 are blocked at that line until p_i executes $FLAG[i] \leftarrow$ *false* (line 10).

   It follows than, from then on, the number of processes competing with p_i to obtain the lock is bounded. A reasoning similar to the used one in Item 1 shows that p_i eventually obtains the lock, which contradicts the initial assumption and concludes the proof of the lemma for that case.

3. Process p_i never exits the loop of line 05. We show that this case cannot occur.

   Let us assume by contradiction that p_i loops forever at line 05. This means that each time it evaluates the predicate at line 05 we have $TURN \neq i \ \wedge \ FLAG[TURN]$. Let $TURN = k_1$ when read by p_i.

   According to Item 1 and Item 2, it follows that p_{k_1} eventually exits the loop line at 05 because it finds $TURN \neq k_1$ (Item 1) or $FLAG[TURN]$ is false (Item 2) and consequently it eventually obtains the lock. Hence p_{k_1} later executes $FLAG[k_1] \leftarrow$ *false* (line 10) and $TURN \leftarrow (k_1 \mod n) + 1$ (line 11). Let k_2 be that process identity. If $k_2 = i$, p_i exits the loop. Hence, let us assume that $k_2 \neq i$. If p_i reads *false* from $FLAG[k_2]$ it stops looping and we are in one of the two previous items.

   If p_i reads always *true* from $FLAG[k_2]$, we are in the same case as previously, replacing k_1 by k_2. We consider then process p_{k_3} such that $k_3 = (k_2 \mod n) + 1$. Etc.

   If follows from the fact that no process is skipped when $TURN$ is modified at line 10 that eventually p_i either is such that $TURN = i$ or reads *false* from $FLAG[k_x]$ for some process identity such that $TURN = k_x$. When this happens we are in the case described in Item 1 or Item 2.

$\square_{Lemma\ 3}$

Theorem 1 *Any invocation of* strong_push() *or* strong_pop() *operation returns a non-$\perp$ value, and all invocations are linearizable. Moreover, the algorithm is contention-sensitive: any* strong_push() *or* strong_pop() *operation invoked in a contention-free context is lock-free and accesses six times the shared memory.*

Proof The fact any strong_push() or strong_pop() operation invoked in a contention-free context is lock-free and accesses six times the shared memory follows directly from the text of the algorithm.

The fact that no operation returns $\perp$ follows from Lemma 1.

All invocations of strong_push() or strong_pop() that return at line 02 trivially terminate. The fact that all other invocations of strong_push() or strong_pop() terminate follows from Lemma 2 and Lemma 3.

The linearization point of a strong_push() (resp., strong_pop()) operation is the linearization point of the last weak_push() (resp., weak_pop()) operation it has executed (as defined in Section 3). $\square_{Theorem\ 1}$

4.4 From a non-blocking lock to a starvation-free lock

When considering Figure 3, let us call starvation_free_lock(i) the code defined by the starred lines 04-06 and starvation_free_unlock(i) the code defined by the starred lines 10-12. The reader can notice that these two operations construct a starvation-free lock from a non-blocking one. The interested reader will find similar constructions in [23, 26].

5 Concluding remarks

Process crashes and unreliable registers When describing the previous algorithms which implement a concurrent task, we have considered that the processes where asynchronous but reliable. The reader can easily verify that these algorithms still work despite process crashes if no process crashes while holding the lock.

We have also assumed that the registers are reliable. Techniques to extend these algorithms to cope with unreliable registers have been studied in several works (e.g., [6]).

Contention managers Contention managers have recently become a hot research topic. The interested reader will find in [4, 25] techniques to extends obstruction-free or non-blocking algorithms to wait-free algorithms (wait-freedom is starvation-freedom in presence of any number of process crashes [7]). She will also find a failure detector-based approach to boost obstruction-freedom or non-blocking to wait-freedom in [5].

More generally, the interested reader will find developments on concurrent objects in [9, 20, 21, 24].

References

[1] Aguilera M.K., Frolund S., Hadzilacos V., Horn S.L. and Toueg S., Abortable and Query-abortable Objects and their Implementations. *Proc. 26th Int'l ACM Symposium on Principles of Distributed Computing (PODC'07)*, pp. 23-32, 2007.

[2] Bernstein Ph.A., Hadzilacos V. and Goodman N., Concurrency Control and Recovery in Database Systems. *Addison Wesley Publishing Company*, 370 pages, 1987.

[3] Dijkstra E.W.D., Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(1):115-138, 1971.

[4] Fich E.F., Luchangco V., Moir M. and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 78-92, 2005.

[5] Guerraoui R., Kapalka M. and Kuznetsov P., The Weakest Failure Detectors to Boost Obstruction-freedom. *Distributed Computing*, 20(6): 415-433, 2008.

[6] Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects: the Case of Atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS LNCS #4671, pp. 47-61, 2007.

[7] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[8] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

[9] Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming, *Morgan Kaufman Pub.*, San Francisco (CA), 508 pages, 2008.

[10] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[11] Hewitt C.E. and Atkinson R.R., Specification and Proof Techniques for Serializers. *IEEE Transactions on Software Engineering*, SE5(1):1-21, 1979.

[12] Hoare C.A.R., Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549-557, 1974.

[13] Jayanti P., Adaptive and Efficient Abortable Mutual Exclusion. *Proc. 22th Int'l ACM Symposium on Principles of Distributed Computing (PODC'03)*, pp. 295-304, 2003.

[14] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

[15] Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101,1986.

[16] Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

[17] Peterson G.L., Myths about Mutual Exclusion. *Information Processing Letters*, 12(3):115-116, 1981.

[18] Raynal M., Algorithms for Mutual Exclusion. *The MIT Press*, ISBN 0-262-18119-3, 107 pages, 1986.

[19] Raynal M., Locks Considered Harmful: a Look at Non-traditional Synchronization. *Proc. 6th Int'l Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS'08)*, Springer Verlag LNCS #5287, pp. 369-380, 2008.

[20] Raynal M., Shared Memory Synchronization in Presence of Failures: an Exercise-based Introduction. *IEEE Int'l Conference on Complex, Intelligent and Software Intensive Systems (CISIS'09)*, IEEE Computer Society Press, pp. 9-18, Fukuoka (Japan), 2009.

[21] Raynal M., On the Implementation of Concurrent Objects. *Tech Report* #1968, IRISA, Université de Rennes (France), 2011. To appear in a Springer Verlag LNCS special issue dedicated to the 75th birthday of Brian Randell.

[22] Shafiei N., Non-blocking Array-based Algorithms for Stacks and Queues. *Proc. th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer Verlag LNCS #5408, pp. 55-66, 2009.

[23] Suzuki I. and Kasami T., A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 3(4):344-349, 1985.

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

[25] Taubenfeld G., Efficient Transformations of Obstruction-free Algorithms into Non-blocking Algorithms. *Proc. 21th Int'l Symposium on Distributed Computing (DISC'07)*, Springer-Verlag LNCS #4731, pp. 450-464, 2007

[26] Taubenfeld G., Contention-Sensitive Data Structure and Algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 157-171, 2009.