Chapter 5
Mutex-Free Concurrent Objects

This chapter is devoted to mutex-free implementations of concurrent objects. Mutex-freedom means that in no way (be it explicit or implicit) is the implementation of a concurrent object allowed to rely on critical sections (locks). The chapter consequently introduces new progress conditions suited to mutex-free object implementations, namely obstruction-freedom, non-blocking, and wait-freedom. It then presents mutex-free implementations of concurrent objects (splitter, queue, stack, etc.) that satisfy these progress conditions. Some of these implementations are based on read/write atomic registers only, while others use also more sophisticated registers that can be accessed by hardware-provided primitive operations such as compare&swap, swap, or fetch&add (which are stronger than base read/write operations). To conclude, this chapter presents an approach based on failure detectors that allows the construction of contention managers that permit a non-blocking or a wait-free implementation of a concurrent object to be obtained from an obstruction-free implementation of that object.

Keywords Contention manager · Implementation boosting · Mutex-freedom · Obstruction-freedom · Non-blocking · Process crash · Progress condition · Wait-freedom

5.1 Mutex-Freedom and Progress Conditions

5.1.1 The Mutex-Freedom Notion

Locks are not always the panacea As we have seen in Chaps. 1 and 3, the systematic use of locks constitutes a relatively simple method to implement atomic concurrent objects defined by total operations. A lock is associated with every object O and all the operation invocations on O are bracketed by acquire_lock() and release_lock() so that at most one operation invocation on O at a time is executed. However, as we are about to see in this chapter, locks are not the only approach to implement atomic objects. Locks

M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*,
DOI: 10.1007/978-3-642-32027-9_5, © Springer-Verlag Berlin Heidelberg 2013

have drawbacks related to process blocking and the granularity of the underlying base objects used in the internal representation of the object under construction.

As far as the granularity of the object protected by a lock is concerned, let us consider a lock-based implementation of a bounded queue object Q with total operations (Q.deq() returns $\perp$ when the queue is empty and Q.enq() returns $\top$ when the queue if full). The use of a single lock on the whole internal representation of the queue prevents Q.enq() and Q.deq() from being executed concurrently. This can decrease the queue efficiency, as nothing prevents these two operations from executing concurrently when the queue is neither empty nor full. A solution consists in using locks at a finer granularity level in order to benefit from concurrency and increase efficiency. Unfortunately this makes deadlock prevention more difficult and, due to their very nature, locks cannot eliminate the blocking problem.

The drawback related to process blocking is more severe. Let us consider a process p that for some reason (e.g., page fault) stops executing during a long period in the middle of an operation on an object O. If we use locks, as we have explained above, the processes which have concurrently invoked an operation on O become blocked until p terminates its own operation. When such a scenario occurs, processes suffer delays due to other processes. Such an implementation is said to be *blocking-prone*. The situation is even worse if the process p crashes while it is in the middle of an operation execution. (In an asynchronous system a crash corresponds to the case where the speed of the corresponding process becomes and remains forever equal to 0, this being never known by the other processes. This point is developed below at the end of Sect. 5.1.2.) When this occurs, p never releases the lock, and consequently, all the processes that will invoke an operation on O will become blocked forever. Hence, the crash of a process creates an infinite delay that can entail a deadlock on all operations accessing the object O.

These observations have motivated the design of concurrent object implementations that do not use locks in one way or another (i.e., explicitly or implicitly). These implementations are called *mutex-free*.

Operation level versus implementation level Let us consider an object O with two operations O.op1() and O.op2(). At the user level, the (correct) behaviors of O are defined by the traces of its sequential specification.

When considering the implementation level, the situation is different. Each execution of O.op1() or O.op2() corresponds to a sequence of invocations of base operations on the base objects that constitute the internal representation of O.

If the implementation of O is lock-based and we do not consider the execution of the base operations that implement acquire_lock() and release_lock(), the sequence of base operations produced by an invocation of O.op1() or O.op2() cannot be interleaved with the sequence of base operations produced by another operation invocation. When the implementation is mutex-free, this is no longer the case, as depicted in Fig. 5.1.

Figure 5.1 shows that the invocations of O.op1() by p_1, O.op2() by p_2, and O.op1() by p_3 are linearized in that order (i.e., they appear to have been executed in that order from an external observer point of view).

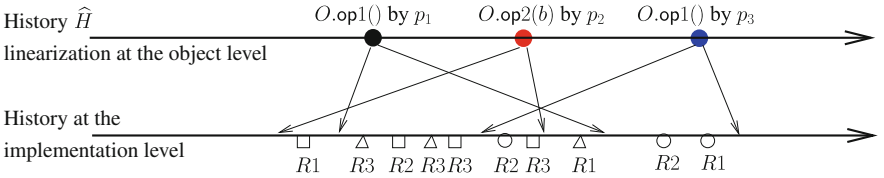

Fig. 5.1 Interleaving at the implementation level

Let us assume that the internal representation of O is made up of three base objects: $R1$, $R2$, and $R3$, which are atomic. It follows from the locality property of the atomicity consistency condition that their invocations are totally ordered (see Chap. 4). In the figure, the ones issued by p_1 are marked by a triangle, the ones issued by p_2 are marked by a square, and the ones issued by p_3 are marked by a circle. The name on the base object accessed by a process appears below the corresponding square, triangle, or circle.

Mutex-free implementation An implementation of an object O is mutex-free if no code inside an operation on O is protected by a critical section. The only atomicity notion that is used by such an implementation is the one on the base operations on the objects which constitute the internal representation of O.

It follows that, inherently, a mutex-free implementation of an object O allows base operations generated by the invocations of operations on O to be interleaved (as depicted in Fig. 5.1). (On the contrary, it is easy to see that the aim of locks is to prevent such interleaving scenarios from occurring.)

In order for a mutex-free implementation to be meaningful, unexpected and arbitrarily long pauses of one or more processes which execute operations on O must not prevent the progress of other processes that invoke operations on the same object O. This observation motivates the definition of progress conditions suited to mutex-free implementations.

5.1.2 Progress Conditions

As shown in Chap. 1, deadlock-freedom and starvation-freedom are the relevant progress conditions when one has to implement a lock object (i.e., solve the mutual exclusion problem) or (by "transitivity") implement a mutex-based atomic object.

Due to the possible interleaving of base operations generated by a mutex-free implementation of a concurrent object, the situation is different from the one encountered in lock-based implementations, and consequently, progress conditions suited to mutex-free implementations must be defined. Going from the weakest to the strongest, this section defines three progress conditions for mutex-free implementations.

Obstruction-freedom *Obstruction-freedom* is a progress condition related to concurrency. An algorithm implementing an operation op() is *obstruction-free* if it satisfies the following property: each time an invocation of op() is executed in

isolation, it does terminate. More generally, an object implementation is obstruction-free if the implementation of each of its operations is obstruction-free.

"*Execute in isolation*" means that there is a point in time after which no other invocation of any operation on the same object is executing. It is nevertheless possible that other invocations of operations on the same object are pending (started and not yet terminated). If this is the case, "*in isolation*" means that these operation invocations have momentarily stopped their execution. From a practical point of view, "*in isolation*" means that a process executes alone during a "*long enough period*". This is because, as the processes are asynchronous, no upper bound on the time needed by a process to execute an operation can be determined. The processes have no notion of time duration, and consequently, the best that can be said is "*long enough period*".

Let us observe that, in the presence of concurrency, it is possible that no invocation on any operation does ever terminate. Let us also observe that nothing prevents a particular obstruction-free implementation from doing more than what is required. (This occurs, for example, when the implementation guarantees termination of an operation in specific scenarios where there are concurrent accesses to the internal representation of the object. In that case, the implementation designer has to specify the additional specific progress property L that is ensured, and the implementation is then the progress condition defined as "obstruction-freedom + L".)

The difficulty in designing an object implementation that is both mutex-free and obstruction-free comes from the fact that the safety properties attached to the internal representation of the objects (usually expressed with invariants) have to be maintained whatever the number of concurrent operation invocations that are modifying this state. In other words, when considering mutex-free object implementations, obstruction-freedom is not given for free.

Non-blocking *Non-blocking* is a stronger progress condition than obstruction-freedom. Its definition involves potentially all the operations of an object (it is not defined for each operation separately). The implementation of an object O is *non-blocking* if, as soon as processes have invoked operations on O, at least one invocation of an operation on O terminates.

As an example let us consider the case where two invocations of Q.enq() and one invocation of Q.deq() are concurrently executing. Non-blocking states that one of them terminates. If no new invocation of Q.enq() or Q.deq() is ever issued, it follows from the non-blocking property that the three previous invocations eventually terminate (because, after one invocation has terminated, the non-blocking property states that one of the two remaining ones eventually terminates, etc.). Differently, if new operation invocations are permanently issued, it is possible that some invocations never terminates.

The non-blocking progress condition is nothing else than deadlock-freedom in the context of mutex-free implementations. While the term "deadlock-freedom" is

associated with lock-based implementations, the term "non-blocking" is used as its counterpart for mutex-free implementations.

Wait-freedom *Wait-freedom* is the strongest progress condition. The algorithm implementing an operation is *wait-free* if it always terminates. More generally, an object implementation is wait-free if any invocation of any of its operations terminates. This means that operation termination is guaranteed whatever the asynchrony and concurrency pattern.

Wait-freedom is nothing else than starvation-freedom in the context of mutex-free implementations. It means that, when a process invokes an object operation, it terminates after having executed a finite number of steps. Wait-freedom can be refined as follows (where the term "step" is used to denote the execution of an operation on an underlying object of the internal representation of the object O):

- Bounded wait-freedom. In this case there is an upper bound on the number of steps that the invoking process has to execute before terminating its operation. This bound may depend on the number of processes, on the size of the internal representation of the object, or both.
- Finite wait-freedom. In this case, there is no bound on the number of steps executed by the invocation of an operation before it terminates. This number is finite but cannot be bounded.

When processes may crash A process crashes when it stops its execution prematurely. Due to the asynchrony assumption on the speed of processes, a crash can be seen as if the corresponding process pauses during an infinitely long period before executing its next step. Asynchrony, combined with the fact that no base shared-memory operation (read, write, compare&swap, etc.) provides processes with information on failures, makes it impossible for a process to know if another process has crashed or is only very slow. It follows that, when we consider mutex-free object implementations, the definition of obstruction-freedom, non-blocking, and wait-freedom copes naturally with any number of process crashes.

Of course, if a process crashes while executing an object operation, it is assumed that this invocation trivially terminates. As we have seen in Chap. 4 devoted to the atomicity concept, this operation invocation is then considered either as entirely executed (and everything appears as if the process crashed just after the invocation) or not at all executed (and everything appears as if the process crashed just before the invocation). This is the *all-or-nothing* semantics associated with crash failures from the atomicity consistency condition point of view.

Hierarchy of progress conditions It is easy to see that obstruction-freedom, non-blocking, and wait-freedom define a hierarchy of progress conditions for mutex-free implementations of concurrent objects.

More generally, the various progress conditions encountered in the implementation of concurrent objects are summarized in Table 5.1.

Table 5.1 Progress conditions for the implementation of concurrent objects

Lock-based implementations	Mutex-free implementations
	Obstruction-freedom
Deadlock-freedom	Non-blocking
Starvation-freedom	Wait-freedom

5.1.3 Non-blocking with Respect to Wait-Freedom

The practical interest of non-blocking object implementations When there are very few conflicts (i.e., it is rare that processes concurrently access the same object), a non-blocking implementation is practically wait-free. This is because, in the very rare occasions where there are conflicting operations, enough time elapses before a new operation is invoked. So, thanks to the non-blocking property, the conflicting invocations have enough time to terminate one after the other.

This observation motivates the design of non-blocking implementations because they are usually more efficient and less difficult to design than wait-free implementations.

The case of one-shot objects A one-shot object is an object accessed at most once by each process. As an example, a one-shot stack is a stack such that any process invokes the operation push() or the operation pop() at most once during an execution.

Theorem 16 *Let us consider a one-shot object accessed by a bounded number of processes. Any non-blocking implementation of such an object is wait-free.*

Proof Let n be the number of processes that access the object. Hence, there are at most n concurrent operation invocations. As the object is non-blocking there is a finite time after which one invocation terminates. There are then at most $(n - 1)$ concurrent invocations, and as the object is non-blocking, one of them terminates, etc. It follows that each operation invocation issued by a correct process eventually terminates. □

5.2 Mutex-Free Concurrent Objects

5.2.1 The Splitter: A Simple Wait-Free Object from Read/Write Registers

Definition The splitter object was implicitly used in Chap. 1 when presenting Lamport's fast mutual exclusion algorithm. A *splitter* is a concurrent object that provides processes with a single operation, denoted direction(). This operation returns a value to the invoking process. The semantics of a splitter is defined by the following properties:

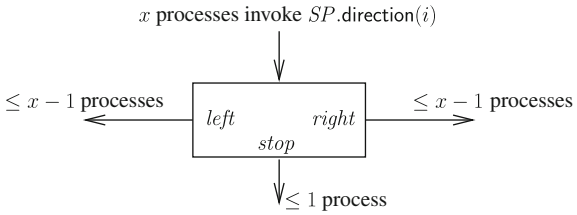

Fig. 5.2 Splitter object

- Validity. The value returned by direction() is *right*, *left*, or *stop*.
- Concurrent execution. If x processes invoke direction(), then:
  - At most $x-1$ processes obtain the value *right*,
  - At most $x-1$ processes obtain the value *left*,
  - At most one process obtains the value *stop*.
- Termination. Any invocation of direction() terminates.

A splitter (Fig. 5.2) ensures that (a) not all the invoking processes go in the same direction, and (b) the direction *stop* is taken by at most one process and exactly one process in a solo execution. As we will see in this chapter, splitters are base objects used to build more sophisticated concurrent objects.

Let us observe that, for $x = 1$, the concurrent execution property becomes: if a single process invokes direction(), only the value *stop* can be returned. This property is sometimes called the "solo execution" property.

A wait-free implementation A very simple wait-free implementation of a splitter object *SP* is described in Fig. 5.3. The internal representation is made up of two MWMR atomic registers: *LAST*, which contains a process index (its initial value is arbitrary), and a binary register *DOOR*, whose domain is {*open*, *closed*} and which is initialized to *open*.

When a process p_i invokes *SP*.direction() it first writes its index i in the atomic register *LAST* (line 1). Then it checks if the door is open (line 2). If the door has been closed by another process, p_i returns *right* (line 3). Otherwise, p_i closes the door (which can be closed by several processes, line 4) and then checks if it was the last process to have invoked direction() (line 5). If this is the case, we have $LAST = i$ and p_i returns *stop*, otherwise it returns *left*.

A process that obtains the value *right* is actually a "late" process: it arrived late at the splitter and found the door closed. Differently, a process p_i that obtains the value *left* is actually a "slow" process: it set $LAST \leftarrow i$ but was not quick enough during the period that started when it wrote its index i into *LAST* (line 1) and ended when it read *LAST* (line 5). According to the previous meanings for "late" and "slow", not all the processes can be late, not all the processes can be slow, and at most one process can be neither late not slow, being "timely" and obtaining the value *stop*.

Theorem 17 *The algorithm described in Fig. 5.3 is a correct wait-free implementation of a splitter.*

```
operation SP.direction(i) is
(1)  LAST ← i;
(2)  if (DOOR = closed)
(3)    then return(right)
(4)    else DOOR ← closed;
(5)         if (LAST = i)
(6)           then return(stop)
(7)           else return(left)
(8)         end if
(9)  end if
end operation.
```

Fig. 5.3 Wait-free implementation of a splitter object (code for process p_i)

Proof The algorithm of Fig. 5.3 is basically the same as the one implementing the operation conc_abort_op() presented in Fig. 2.12 (Chap. 2); $abort_1$, $abort_2$, and *commit* are replaced by *right*, *left*, and *stop*. The following proof is consequently very close to the proof of Theorem 4. We adapt and repeat it here for self-containment of the chapter.

The validity property follows trivially from the fact that the only values that can be returned are *right* (line 3), *stop* (line 6), and *left* (line 7).

As far as the termination property is concerned, let us observe that the code of the algorithm contains neither loops nor **wait** statements. It follows that any invocation of *SP*.direction() by a process (which does not crash) does terminate and returns a value. The implementation is consequently wait-free.

As far as the solo execution property is concerned, it follows from a simple examination of the code and the fact that the door is initially open that, if a single process invokes *SP*.direction() (and does not crash before executing line 6), it returns the value *stop*.

Let us now consider the concurrent execution property. For a process to obtain *right*, the door must be closed (lines 2–3). As the door is initially open, it follows that the door was closed by at least one process p and this was done at line 4 (which is the only place where a process can close the door). According to the value of *LAST* (line 5), process p will return *stop* or *left*. It follows that, among the x processes which invoke *SP*.direction(), at least one does not return the value *right*.

As far as the value *left* is concerned, we have the following. Let p_i be the last process that writes its index i into the register *LAST* (as this register is atomic, the notion of "last" writer is well defined). If the door is closed, it obtains the value *right*. If the door is open, it finds $LAST = i$ and obtains the value *stop*. Hence, not all processes can return *left*.

Let us finally consider the value *stop*. Let p_i be the first process that finds *LAST* equal to its own index i (line 5). This means that no process p_j, $j \neq i$, has modified *LAST* during the period starting when it was written by p_i at line 1 and ending when it was read by p_i at line 5 (Fig. 5.4). It follows that any process p_j that modifies *LAST*

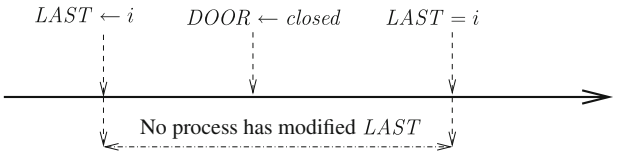

Fig. 5.4 On the modification of *LAST*

after this register was read by p_i will find the door closed (line 2). Consequently, any such p_j cannot obtain the value *stop*. □

The reader may check that the proof of the splitter object remains valid if processes crash.

5.2.2 A Simple Obstruction-Free Object from Read/Write Registers

This section presents a simple obstruction-free timestamp object built from atomic registers. Actually, the object is built from splitters, which as we have just seen, are in turn built from atomic read/write registers.

Definition The object is a weak timestamp generator object which provides the processes with a single operation denoted get_timestamp() which returns an natural integer. Its specification is the following:

- Validity. No two invocations of get_timestamp() return the same value.
- Consistency. Let $\mathsf{gt}_1()$ and $\mathsf{gt}_2()$ be two distinct invocations of get_timestamp(). If $\mathsf{gt}_1()$ returns before $\mathsf{gt}_2()$ starts, the timestamp returned by $\mathsf{gt}_2()$ is greater than the one returned by $\mathsf{gt}_1()$.
- Termination. Obstruction-freedom.

It is easy to see that a lock-based implementation of a timestamp object is trivial: an atomic register protected by a lock is used to supply timestamps. But, as already noticed, locking and obstruction-freedom are incompatible in asynchronous crash-prone systems. It is also trivial to implement this object directly from the fetch&add() primitive. The presentation of such a timestamp generator object is mainly pedagogic, namely showing an obstruction-free implementation built on top of read/write registers only.

An algorithm The obstruction-free implementation relies on the following underlying data structures:

- *NEXT* defines the value of the next integer that can be used as a timestamp. It is initialized to 1.
- *LAST* is an unbounded array of atomic registers. A process p_i deposits its index i in $LAST[k]$ to indicate it is trying to obtain the timestamp k.

- *COMP* is another unbounded array of atomic Boolean registers with each entry initialized to *false*. A process p_i sets $COMP[k]$ to *true* to indicate that it is competing for the timestamp k (hence several processes can write *true* into $COMP[k]$). For any k, $COMP[k]$ is initialized to *false*.

The algorithm implementing the obstruction-free operation get_timestamp() is described in Fig. 5.5. It is inspired by the wait-free algorithm described in Fig. 5.3 that implements a splitter. (The pair of registers $LAST[k]$ and $COMP[k]$ in Fig. 5.5 plays the same role as the registers *LAST* and *CLOSED* in Fig. 5.3.) A process p_i first reads the next possible timestamp value (register *NEXT*). Then it enters a loop that it will exit after it has obtained a timestamp (line 6).

In the loop, p_i first writes its index in $LAST[k]$ to indicate that it is the last process competing for the timestamp k (line 3). Then, if it finds $COMP[k] = false$, p_i sets it to *true* to indicate that at least one process is competing for the timestamp k. Let us observe that it is possible that several processes find $COMP[k]$ equal to *false* and set it to *true* (lines 4–5). Then, p_i checks the predicate $LAST[k] = i$. If this predicate is satisfied, p_i can conclude that it is the last process that wrote into $LAST[k]$. Consequently, all other processes (if any) competing for timestamp k will find $COMP[k] = true$, and will directly proceed to line 8 to try to obtain timestamp $k + 1$. Hence, they do not execute lines 5–6.

It is easy to see that if, after some time, a single process keeps on executing the algorithm implementing get_timestamp(), it eventually obtains a timestamp. In contrast, when several processes find $COMP[k]$ equal to *false*, there is no guarantee that one of them obtains the timestamp k.

The proof of this mutex-free implementation based on atomic read/write registers only is similar to the proof of a splitter. It is left to the reader. (The fact that, for any timestamp value k, there is at most one process that obtains that value follows from the fact that exactly one splitter is associated with each possible timestamp value.)

```
operation get_timestamp(i) is
(1)  k ← NEXT;
(2)  repeat forever
(3)     LAST[k] ← i;
(4)     if (¬COMP[k])
(5)        then COMP[k] ← true;
(6)             if (LAST[k] = i) then NEXT ← NEXT + 1; return(k) end if
(7)     end if;
(8)  k ← k + 1
(9)  end repeat
end operation.
```

Fig. 5.5 Obstruction-free implementation of a timestamp object (code for p_i)

5.2.3 A Remark on Compare&Swap: The ABA Problem

Definition (**reminder**) The compare&swap operation was introduced in Chap. 2. It is an atomic conditional write provided at hardware level by some machines. As we have seen, its effect can be described as follows. X is the register on which this machine instruction is applied and *old* and *new* two values. The new value *new* is written into X if and only if the actual value of X is *old*. A Boolean result indicates if the write was successful or not.

X.compare&swap(*old*, *new*) **is**
if $(X = old)$ **then** $X \leftarrow new$; return(*true*) **else** return(*false*) **end if**.

The ABA problem When using compare&swap(), a process p_i usually does the following. It first reads the atomic register X (obtaining the value a), then executes statements (possibly involving accesses to the shared memory) and finally updates X to a new value c only if X has not been modified by another process since it was read by p_i. To that end, p_i invokes X.compare&swap(a, c) (Fig. 5.6).

Unfortunately, the fact that this invocation returns *true* to p_i does not allow p_i 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.compare&swap(a, c) both issued by p_i, X could have been updated twice, first by a process p_j that successfully invoked X.compare&swap(a, b) and then by a process p_k that has successfully invoked X.compare&swap(b, a), thereby restoring the value a to X. This is called the ABA problem.

Solving the ABA problem This problem can be solved by associating tags (sequence numbers) with each value that is written. The atomic register X is then composed of two fields $\langle content, tag \rangle$. When it reads X, a process p_i obtains a pair $\langle x, y \rangle$ (where x is the current "data value" of X) and it later invokes X.compare&swap$(\langle x, y \rangle, \langle c, y + 1 \rangle)$ to write a new value c into X. It is easy to see that the write succeeds only if X has continuously been equal to $\langle x, y \rangle$.

statements;
$x \leftarrow X$;
statements possibly involving accesses to the shared memory;
if X.compare&swap(x, c) **then** statements **else** statements **if**;
statements.

Fig. 5.6 A typical use of compare&swap() by a process

5.2.4 A Non-blocking Queue Based on Read/Write Registers and Compare&Swap

This section presents a non-blocking mutex-free implementation of a queue Q due to M. Michael and M. Scott (1996). Interestingly, this implementation was included in the standard Java Concurrency Package. Let us remember that, to be non-blocking, this implementation has to ensure that, in any concurrency pattern, at least one invocation always terminates.

Internal representation of the queue Q The queue is implemented by a linked list as described in Fig. 5.7. The core of the implementation consists then in handling pointers with the help of the compare&swap() primitive.

As far as registers containing pointer values are concerned, the following notations are employed. If P is a pointer register, $P\downarrow$ denotes the object pointed to by P. If X is an object, $\uparrow X$ denotes a pointer that points to X. Hence, $(\uparrow X)\downarrow$ and X denote the same object.

The list is accessed from an atomic register Q that contains a pointer to a record made up of two fields denoted *head* and *tail*. Each of these field is an atomic register.

Each atomic register $(Q\downarrow).head$ and $(Q\downarrow).tail$ has two fields denoted *ptr* and *tag*. The field *ptr* contains a pointer, while the field *tag* contains an integer (see below). To simplify the exposition, it is assumed that each field *ptr* and *tag* can be read independently.

The list is made up of cells such that the first cell is pointed to by $(Q\downarrow).head.ptr$ and the last cell of the list is pointed to by $(Q\downarrow).tail.ptr$.

Let *CELL* be a cell. It is a record composed of two atomic registers. The atomic register $CELL.value$ contains a value enqueued by a process, while (similarly to $(Q\downarrow).head$ and $(Q\downarrow).tail$) the atomic register $CELL.next$ is made up of two fields: $CELL.next.ptr$ is a pointer to the next cell of the list (or $\perp$ if *CELL* is the last cell of the list), and $CELL.next.tag$ is an integer.

Initially the queue contains no element. At the implementation level, the list Q contains then a dummy cell *CELL* (see Fig. 5.8). This cell is such that $CELL.next.ptr$ is (always) irrelevant and $CELL.next.ptr = \perp$. This dummy cell allows for a simpler algorithm. It always belongs to the list and $(Q\downarrow).head.ptr$ always points to it.

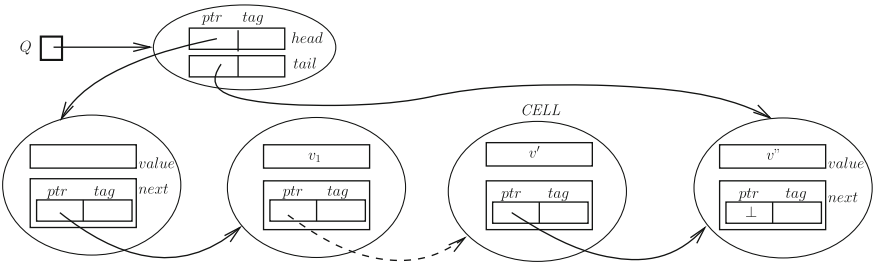

Fig. 5.7 The list implementing the queue

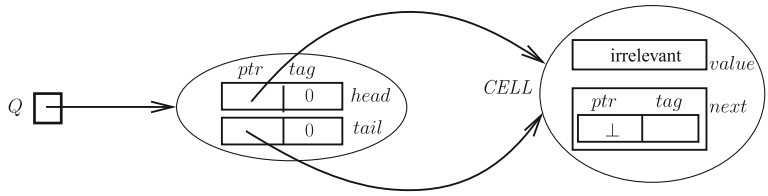

Fig. 5.8 Initial state of the list

Differently, $(Q\downarrow).tail.ptr$ points to the dummy cell only when the list is empty. Moreover, we have initially $(Q\downarrow).head.tag = (Q\downarrow).tail.tag = 0$.

It is assumed that the operation new_cell() creates a new cell in the shared memory, while the operation free_cell(pt) frees the cell pointed to by pt.

The algorithm implementing the operation Q.enq() As already indicated, these algorithms consist in handling pointers in an appropriate way. An interesting point is the fact that they require processes to help other processes terminate their operations. Actually, this helping mechanism is the mechanism that implements the non-blocking property.

The algorithm implementing the enq() operation is described at lines 1–13 of Fig. 5.9. The invoking process p_i first creates a new cell in the shared memory, assigns its address to the local pointer pt_cell, and updates its fields $value$ and $next.ptr$ (line 1). Then p_i enters a loop that it will exit when the value v will be enqueued.

In the loop, p_i executes the following statements. It is important to notice that, in order to obtain consistent pointer values, these statements include sequences of read and re-read (with compare&swap) to check that pointer values have not been modified.

- Process p_i first makes local copies (kept in $\ell tail$ and $\ell next$) of $(Q\downarrow).tail$ and $(\ell tail.ptr\downarrow).next$, respectively. These values inform p_i on the current state of the tail of the queue (lines 3–4).
- Then p_i checks if the content of $(Q\downarrow).tail$ has changed since it read it (line 5). If it has changed, $\ell tail.ptr$ no longer points to the last element of the queue. Consequently, p_i starts the loop again.
- If $\ell tail = (Q\downarrow).tail$ (line 6), p_i optimistically considers that no other process is currently trying to enqueue a value. It then checks if $\ell next.ptr$ is equal to $\perp$.
  - If $\ell next.ptr = \perp$, p_i optimistically considers that $\ell tail$ points to the last element of the queue. It consequently tries to add the new element v to the list (lines 7–8). This is done in two steps, each based on a compare&swap: the first to append the cell to the list, and the second to update the pointer $(Q\downarrow).tail$.
    - Process p_i tries first to append its new cell to the list. This is done by executing the statement $((\ell tail.ptr\downarrow).next)$.compare&swap$(\ell next, \langle \ell cell, \ell next.tag + 1\rangle)$ (line 7). If p_i does not succeed, this is because another process succeeded in appending a new cell to the list. If this is the case, p_i continues looping.

```
operation enq(v) is
(1)   pt_cell ←↑ new_cell(); (pt_cell ↓).value ← v; (pt_cell ↓).next.ptr ← ⊥;
(2)   repeat forever
(3)     ℓtail ← (Q ↓).tail;
(4)     ℓnext ← (ℓtail.ptr ↓).next;
(5)     if (ℓtail = (Q ↓).tail) then
(6)       if (ℓnext.ptr = ⊥)
(7)         then if ((ℓtail.ptr ↓).next).compare&swap(ℓnext, ⟨pt_cell, ℓnext.tag + 1⟩)
(8)                 then((Q ↓).tail).compare&swap(ℓtail, ⟨pt_cell, ℓtail.tag + 1⟩); return(ok)
(9)              end if
(10)        else ((Q ↓).tail).compare&swap(ℓtail, ⟨ℓnext.ptr, ℓtail.tag + 1⟩)
(11)      end if
(12)    end if
(13)  end repeat
end operation.

operation deq() is
(14)  repeat forever
(15)    ℓhead ← (Q ↓).head;
(16)    ℓtail ← (Q ↓).tail;
(17)    ℓnext ← (ℓhead.ptr ↓).next;
(18)    if (ℓhead = (Q ↓).head) then
(19)      if (ℓhead.ptr = ℓtail.ptr)
(20)        then if (ℓnext.ptr = ⊥) then return(empty) end if;
(21)             ((Q ↓).tail).compare&swap(ℓtail, ⟨ℓnext.ptr, ℓtail.tag + 1⟩)
(22)        else result ← (ℓnext.ptr ↓).value;
(23)             if ((Q ↓).head).compare&swap(ℓhead, ⟨ℓnext.ptr, ℓhead.tag + 1⟩)
(24)               then free(ℓhead.ptr); return(result)
(25)             end if
(26)      end if
(27)    end if
(28)  end repeat
end operation.
```

Fig. 5.9 A non-blocking implementation of a queue

- ∗ If process p_i succeeds in appending its new cell to the list, it tries to update the content of $(Q\downarrow).tail$. This is done by executing $(Q\downarrow).tail$).compare&swap ($\ell tail$, $\langle \ell cell, \ell tail.tag + 1\rangle$) (line 8). Finally, p_i returns from its invocation.

  Let us observe that it is possible that the second compare&swap does not succeed. This is the case when, due to asynchrony, another process p_j did the work for p_i by executing line 10 of enq() or line 21 of deq().

– If $\ell next.ptr \neq \bot$, p_i discovers that $\ell next$ does not point to the last element of the queue. Hence, p_i discovers that the value of $(Q\downarrow).tail$ was not up to date when it read it. Another process has added an element to the queue but had not yet updated $(Q\downarrow).tail$ when p_i read it. In that case, p_i tries to help the other process terminate the update of $(Q\downarrow).tail$ if not yet done. To that end, it executes the statement $((Q\downarrow).tail)$.compare&swap($\ell tail$, $\langle \ell next.ptr, \ell tail.tag+1\rangle$) (line 10) before restarting the loop.

Linearization point of Q.enq() The linearization point associated with an enq() operation corresponds to the execution of the compare&swap statement of line 7. This means that an enq() operation appears as if it was executed atomically when the new cell is linked to the last cell of the list.

The algorithm implementing the operation Q.deq() The algorithm implementing the deq() operation is described in lines 14–28 of Fig. 5.9. The invoking process loops until it returns a value at line 24. Due to its strong similarity with the algorithm implementing the enq() operation, the deq() algorithm is not described in detail.

Let us notice that,, if $\ell head \neq (Q\downarrow).head$ (i.e., the predicate at line 18 is false), the head of the list has been modified while p_i was trying to dequeue an element. In that case, p_i restarts the loop.

If $\ell head = (Q\downarrow).head$ (line 18) then the values kept in $\ell head$ and $\ell next$ defining the head of the list are consistent. Process p_i then checks if $\ell head.ptr = \ell tail.ptr$, i.e., if (according to the values it has read at lines 15–16) the list currently consists of a single cell (line 19). If this is the case and this cell is the dummy cell (as witnessed by the predicate $\ell next.ptr = \perp$), the value *empty* is returned (line 20). In contrast, if $\ell next.ptr \neq \perp$, a process is concurrently adding a new cell to the list. To help it terminate its operation, p_i executes $((Q\downarrow).tail)$.compare&swap$(\ell tail, \langle \ell next.ptr, \ell tail.tag + 1\rangle)$ (line 21).

Otherwise ($\ell head \neq (Q\downarrow).head$), there is at least one cell in addition to the dummy cell. This cell is pointed to by $\ell next.ptr$. The value kept in that cell can be returned (lines 22–24) if p_i succeeds in updating the atomic register $(Q\downarrow).head$ that defines the head of the list. This is done by the statement $((Q\downarrow).head)$.compare&swap$(\ell head, \langle \ell next.ptr, \ell head.tag + 1\rangle)$ (line 23). If this compare&swap succeeds, p_i returns the appropriate value and frees the cell (pointed to by $\ell next.ptr$ which was suppressed from the list, line 24). Let us observe that the cell that is freed is the previous dummy cell while the cell containing the returned value v is the new dummy cell.

Linearization point of Q.deq() The linearization point associated with a deq() operation is the execution of the compare&swap statement of line 23 that terminates successfully. This means that a deq() operation appears as if it was executed atomically when the pointer to the head of the list $(Q\downarrow).head$ is modified.

Remarks Both linearization points correspond to the execution of successful compare&swap statements. The two other invocations of compare&swap statements (lines 10 and 21) constitute the helping mechanism that realizes the non-blocking property.

It is important to notice that, due to the helping mechanism, the crash of a process does not annihilate the non-blocking property. If processes crash at any point while executing enq() or deq() operations, at least one process that does not crash while executing its operation terminates it.

5.2.5 A Non-blocking Stack Based on Compare&Swap Registers

The stack and its operations The stack has two operations, denoted push(v) (where v is the value to be added at the top of the stack) and pop(). It is a bounded stack: it can contain at most k values. If the stack is full, push(v) returns the control value *full*, otherwise v is added to the top of the stack and the control value *done* is returned. The operation pop() returns the value that is at the top of the stack (and suppresses it from the stack), or the control value *empty* if the stack is empty.

Internal representation of the stack This non-blocking implementation of an atomic stack is due to N. Shafiei (2009). The stack is implemented with an atomic register denoted *TOP* and an array of $k+1$ atomic registers denoted $STACK[0..k]$. These registers can be read and can be modified only by using the compare&swap() primitive.

- *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: the field $STACK[x].val$, which contains a value, and the field $STACK[x].sn$, which contains a sequence number (used to prevent the ABA problem as far as $STACK[x]$ is concerned).

  $STACK[0]$ is a dummy entry initialized to $\langle \perp, -1\rangle$. Its first field always contains the default value $\perp$. As far as the other entries are concerned, $STACK[x]$ $(1 \leq x \leq k)$ is initialized to $\langle \perp, 0\rangle$.

The array *STACK* is used to store the contents 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 contents of *TOP* and $STACK[x]$ are modified with the help of the conditional write instruction compare&swap() (which 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 leaves the corresponding effective 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 (as far as the internal representation of the stack is concerned).

The algorithm implementing the operation push(v) When a process p_i invokes push(v), it enters a **repeat** loop that it will exit at line 4 or line 7. The process first reads the content of *TOP* (which contains the last operation on the stack) and stores its three fields in its local variables *index*, *value*, and *seqnb* (line 2).

Then, p_i calls the internal procedure help(*index*, *value*, *seqnb*) to help terminate the previous stack operation (line 3). That stack operation (be it a push() or a pop()) is required to write the pair $\langle value, seqnb\rangle$ into $STACK[index]$. To that end, p_i invokes $STACK[index]$.compare&swap.(old, new) with the appropriate values *old* and *new* so that the write is executed only if not yet done (lines 17–18).

After its help (which 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 4). If the stack is not full, it tries to modify *TOP* so that it registers its push operation. This invocation of *TOP*.compare&swap() (line 7) succeeds if no other process has modified *TOP* since it was read by p_i at line 2. If it succeeds, *TOP* takes its new value and push(v) returns the control value *done* (lines 7). Otherwise p_i executes the body of the **repeat** loop again until its invocation of push() succeeds.

The triple of values to be written in *TOP* at line 7 is computed at lines 5–6. 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* = ⟨*index* + 1, v, *sn_of_next* + 1⟩, 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 algorithm implementing the operation 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 the push() **and** pop() **operations** The operations that terminate are linearizable; i.e., they can be totally ordered on the time line, each operation being associated with a single point of that line after its start event and before its end event. Its start event corresponds to the execution of the first statement of an operation, and its end event corresponds to the execution of the return() statement. More precisely, an invocation of an operation appears as if it was atomically executed

- when it reads *TOP* (at line 2 or 10) if it returns *full* or *empty* (at line 4 or 12),
- or at the time at which its invocation *TOP*.compare&swap$(-, -)$ (at line 7 or 15) is successful (i.e., returns *true*).

Theorem 18 *The stack implementation described in Fig. 5.10 is non-blocking.*

Proof Let us first observe that, if a process p executes an operation while no other process executes an operation, it does terminate. This is because the triple (*index*, *value*, *seqnb*) it has read from *TOP* at line 2 or 11 is still in *TOP* when it executes *TOP*.compare&swap(⟨*index*, *value*, *seqnb*⟩, *newtop*) at line 7 or 15. Hence, the compare&swap() is successful and returns the value *true*, and the operation terminates.

Let us now consider the case where the invocation of an operation by a process p does not terminate (while p does not crash). This means that, between the read of *TOP* at line 2 (or line 11) and the conditional write *TOP*.compare&swap(⟨*index*, *value*, *seqnb*⟩, *newtop*) at line 7 (or line 15) issued by p, the atomic register *TOP* was modified. According to the code of the push() and pop() operations, the only statement that modifies *TOP* is the compare&swap() issued at line 7 (or line 15). If follows that another invocation of compare&swap() was successful, which means that another push() and pop() terminated, completing the proof of the non-blocking property. □

```
operation push(v) is
(1)  repeat forever
(2)     (index, value, seqnb) ← TOP;
(3)     help(index, value, seqnb);
(4)     if (index = k) then return(full) end if;
(5)     sn_of_next ← STACK[index + 1].sn;
(6)     newtop ← ⟨index + 1, v, sn_of_next + 1⟩;
(7)     if TOP.compare&swap(⟨index, value, seqnb⟩, newtop) then return(done) end if
(8)  end repeat
end operation.

operation pop() is
(9)  repeat forever
(10)    (index, value, seqnb) ← TOP;
(11)    help(index, value, seqnb);
(12)    if (index = 0) then return(empty) end if;
(13)    belowtop ← STACK[index − 1];
(14)    newtop ← ⟨index − 1, belowtop.val, belowtop.sn + 1⟩;
(15)    if TOP.compare&swap(⟨index, value, seqnb⟩, newtop) then return(value) end if
(16) end repeat
end operation.

procedure help(index, value, seqnb):
(17) stacktop ← STACK[index].val;
(18) STACK[index].compare&swap(⟨stacktop, seqnb − 1⟩, ⟨value, seqnb⟩)
end procedure.
```

Fig. 5.10 Shafiei's non-blocking atomic stack

5.2.6 A Wait-Free Stack Based on Fetch&Add and Swap Registers

The non-blocking implementation of a stack presented in the previous section was based on a bounded array of compare&swap atomic registers. This section presents a simple wait-free implementation of an unbounded stack. This construction, which is due to Y. Afek, E. Gafni, and A. Morrison (2007), uses a fetch&add register and an unbounded array of swap registers.

Internal representation of the stack *STACK* This representation is made up of the following atomic registers which are not base read/write registers:

- *REG*[0..∞) is an array of atomic registers which contains the elements of the stack. Each *REG*[x] can be written by any process. It can also be accessed by any process by invoking the primitive *REG*[x].swap(v), which writes atomically v into *REG*[x] and returns its previous value. Initially each *REG*[x] is initialized to a default value $\perp$ (which remains always unknown to the processes).

  REG[0] contains always the value $\perp$ (it is used only to simplify the description of the algorithm).

- *NEXT* is an atomic register that contains the index of the next entry where a value can be deposited. It is initialized to 1. This register can be read by any process. It can be modified by any process by invoking *NEXT*.fetch&add(), which adds 1 to *NEXT* and returns its new value.

The algorithm implementing the operation push(v) This algorithm, described in Fig. 5.11, is simple. When a process invokes *STACK*.enq(v), it first computes the next free entry (*in*) of the array (line 1) and then deposits its value in *REG*[*in*] (line 2).

The algorithm implementing the operation pop() The code of this algorithm appears in Fig. 5.11. When it invokes *STACK*.pop(), a process p_i first determines the last entry (*last*) in which a value has been deposited (line 4). Then, starting from *REG*[*last*], p_i scans the array *REG*[0..*last*] (line 5). It stops scanning (downwards) at the first register *REG*[x] whose value is different from $\perp$ and returns it (lines 6–7). Let us notice that, if p_i returns a value, it has previously suppressed it from the corresponding register when it invoked *REG*[x].swap($\perp$). If the scan does not allow p_i to return a value, the queue is empty and, accordingly, p_i executes return(*empty*) (line 9).

A remark on process crashes As indicated previously in this chapter, any mutex-free algorithm copes naturally with process crashes. As the base operations that access the shared memory (at the implementation level) are atomic, a process crashes before or after such a base operation.

To illustrate this point, let us first consider a process p_i that crashes while it is executing *STACK*.push(v). There are two cases:

- Case 1: p_i crashes after it has executed the atomic statement *REG*[*in*] $\leftarrow v$ (line 2). In this case, from an external observer point of view, everything appears as if p_i crashed after it invoked *STACK*.push(v).
- Case 2: p_i crashes after it has obtained an index value (line 1) and before it invokes the atomic statement *REG*[*in*] $\leftarrow v$. In this case, p_i has obtained an entry *in* from

```
operation push(v) is
(1)  in ← NEXT.fetch&add() − 1;
(2)  REG[in] ← v;
(3)  return()
end operation.

operation Q.pop() is
(4)  last ← NEXT − 1;
(5)  for x from last to 0 do
(6)      aux ← REG[x].swap(⊥);
(7)      if (aux ≠ ⊥) then return(aux) end if
(8)  end for,
(9)  return(empty)
end operation.
```

Fig. 5.11 A simple wait-free implementation of an atomic stack

NEXT but did not deposit a value into $REG[in]$, which consequently will remain forever equal to $\perp$. In this case, from an external observer point of view, everything appears as if the process crashed before invoking $STACK.\mathsf{push}(v)$.

From an internal point of view, the crash of p_i just before executing $REG[in] \leftarrow v$ entails an increase of *NEXT*. But as the corresponding entry of the array *REG* will remain forever equal to $\perp$, this increase of *NEXT* can only increase the duration of the loop but cannot affect its output.

Let us now consider a process p_i that crashes while it is executing $STACK.\mathsf{pop}()$. If p_i crashes after it has executed the statement $aux \leftarrow REG[x].\mathsf{swap}(\perp)$ (line 6), which has returned it a value, everything appears to an external observer as if p_i crashed after the invocation of $STACK.\mathsf{pop}()$. In the other case, everything appears to an external observer as if p_i crashed before the invocation of $STACK.\mathsf{pop}()$.

Wait-freedom versus bounded wait-freedom A simple examination of the code of push() shows that this operation is bounded wait-free: it has no loop and accesses the shared memory twice.

In contrast, while all executions of $STACK.\mathsf{pop}()$ terminate, none of them can be bounded. This is because the number of times the loop body is executed depends on the current value of *NEXT*, which may increase forever. Hence, while no execution of pop() loops forever, there is no bound on the number of iterations an execution of $Q.\mathsf{pop}()$ has to execute. Hence, the algorithm implementing the operation pop() is wait-free but not bounded wait-free.

On the linearization points of push() **and** pop() It is important to notice that the linearization points of the invocations of push() and pop() cannot be statically defined in a deterministic way. They depend on race conditions which occur during the execution. This is due to the non-determinism inherent to each concurrent computation.

As an example, let us consider the stack described in Fig. 5.12. The values a, b, d, e, f, and g have been written into *REG* at the indicated entries. A process p_i which has invoked $\mathsf{push}(c)$ obtained the index value x (at line 1) before the invocations of $\mathsf{push}(d)$, etc., $\mathsf{push}(g)$ obtained the indexes $(x+1)$, $(x+2)$, $(x+4)$, and $(x+5)$, respectively. (The index $(x+3)$ obtained by a process that crashed just after it obtained that index value.) Moreover, p_i executes $REG[x] \leftarrow c$ only after d, e, f, and g have been written into the array *REG* and the corresponding invocations of push() have terminated. In that case the linearization point associated with $\mathsf{push}(c)$ is not the time at which it executes $REG[x] \leftarrow c$ but a time instant just before the linearization points associated with $\mathsf{push}(d)$.

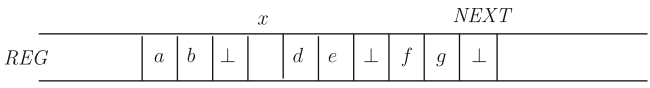

Fig. 5.12 On the linearization points of the wait-free stack

If p_i executes $REG[x] \leftarrow c$ after all the values deposited at entries with an index greater than x have been removed from the stack, and before new values are pushed onto the stack, then the linearization point associated with push(c) is the time at which p_i executes $REG[x] \leftarrow c$.

While the definition of the linearization points associated with the operation invocations on a concurrent object is sometimes fairly easy, the previous wait-free implementation of a stack (whose algorithms are simple) shows that this is not always the case. This is due to the net effect of the mutex-freedom requirement and asynchrony.

5.3 Boosting Obstruction-Freedom to Stronger Progress in the Read/Write Model

Let us consider the case where (a) the processes can cooperate by accessing base read/write atomic registers only and (b) any number of processes may crash. Let us suppose that, in such a context, we have an obstruction-free implementation of a concurrent object (hence this implementation relies only on read/write atomic registers). An important question is then the following: Is it possible to boost this implementation in order to obtain a non-blocking or even a wait-free implementation? This section presents an approach based on failure detectors that answers this question.

5.3.1 Failure Detectors

As already indicated, given an execution E, a process is said to be *correct* in E if it does not crash in execution E. Otherwise, it is faulty in E.

A failure detector is a device (object) that provides each process with a read-only variable that contains information related to failures. According to the type and the quality of this information, several types of failure detector can be defined. Two types of failure detector are defined below.

When considering two failure detectors, one can be stronger than the other or they can be incomparable. Failure detector $FD1$ is *stronger* than failure detector $FD2$ if there is an algorithm that builds $FD2$ from $FD1$ and atomic read/write registers. If additionally $FD1$ cannot be built from $FD2$, then $FD1$ is *strictly stronger* than $FD2$. If $FD1$ is stronger than $FD2$ and $FD2$ is stronger than $FD1$, then $FD1$ and $FD2$ have the same computability power in the sense that the information on failures provided by either of them can be obtained from the other. If two failures detectors are such that neither of them is stronger than the other one, they are *incomparable*.

The failure detector Ω_X (eventually restricted leadership) Let X be any non-empty subset of process indexes. The failure detector denoted Ω_X provides each

process p_i with a local variable denoted $ev\_leader(X)$ (eventual leader in the set X) such that the following properties are always satisfied:

- Validity. At any time, the variable $ev\_leader(X)$ of any process contains a process index.
- Eventual leadership. There is a finite time after which the local variables $ev\_leader(X)$ of the correct processes of X contain the same index, which is the index of one of them.

This means that there is an arbitrarily long anarchy period during which the content of any local variable $ev\_leader(X)$ can change and, at the same time, distinct processes can have different values in their local variables. However, this anarchy period terminates for the correct processes of X, and when it has terminated, the local variable $ev\_leader(X)$ of the correct processes of X contain forever the same index, and it is the index of one of them. The time at which this occurs is finite but remains unknown to the processes. This means that, when a process of X reads x from $ev\_leader(X)$, it can never be sure that p_x is correct. In that sense, the information on failures (or the absence of failures) provided by Ω_X is particularly weak.

Remark on the use of Ω_X This failure detector is usually used in a context where X denotes a dynamically defined subset of processes. It then allows these processes to rely on the fact that one of them (which is correct) is eventually elected as their common leader.

It is possible that, at some time, a process perceived locally X as being x_i while another process p_j perceives it as being $x_j \neq x_i$. Consequently, the local read-only variables provided by Ω_X are denoted $ev\_leader(x_i)$ at p_i and $ev\_leader(x_j)$ at p_j. As x_i and x_j may change with time, this means that Ω_X may potentially be required to produce outputs for any non-empty subset x of Π (the whole set of processes composing the system).

The failure detector $\Diamond P$ (eventually perfect) This failure detector provides each process p_i with a local set variable denoted *suspected* such that the following properties are always satisfied:

- Eventual completeness. Eventually the set $suspected_i$ of each correct process p_i contains the indexes of all crashed processes.
- Eventual accuracy. Eventually the set $suspected_i$ of each correct process p_i contains only indexes of crashed processes.

As with Ω_X (a) there is an arbitrary long anarchy period during which each set $suspected_i$ can contain arbitrary values, and (b) the time at which this anarchy period terminates remains unknown to the processes.

It is easy to see that $\Diamond P$ is stronger than Ω_X (actually, it is strictly stronger). Let assume that we are given $\Diamond P$. The output of Ω_X can be constructed as follows. For a process p_i such that $i \notin X$ the current value of $ev\_leader(X)$ is any process index and it can change at any time. For a process p_i such that $i \in X$, the output of Ω_X is

defined as follows: $ev\_leader(X) = \min\big((\Pi \setminus suspected) \cap X\big)$. The reader can check that the local variables $ev\_leader(X)$ satisfy the validity and eventual leadership of Ω_X.

5.3.2 Contention Managers for Obstruction-Free Object Implementations

A *contention manager* is an object whose aim is to improve the progress of processes by providing them with contention-restricted periods during which they can complete object operations.

As we consider obstruction-free object implementations, the idea is to associate a contention manager with each obstruction-free implementation. Hence, the role of a contention manager is to create "favorable circumstances" so that object operations execute without contention in order to guarantee their termination. For these "favorable circumstances", the contention manager uses the computational power supplied by a failure detector.

The structure of the boosting is described in Fig. 5.13. A failure-detector-based manager implements two (control) operations denoted need_help() and stop_help() which are used by the obstruction-free implementation of an object as follows:

- need_help() is invoked by a process which is executing an object operation to inform the contention manager that it has detected contention and, consequently, needs help to terminate its operation invocation.
- stop_help() is invoked by a process to inform the contention manager that it terminates its current operation invocation and, consequently, no longer needs help.

As an example let us consider the timestamping object defined in Sect. 5.2.2 whose obstruction-free implementation is given in Fig. 5.5. The enrichment of this implementation to benefit from contention manager boosting appears in Fig. 5.14. The invocations of need_help() and stop_help() are underlined. As we can see, need_help() is invoked by a process p_i when it discovers that there is contention on the timestamp k and it decides accordingly to proceed to $k + 1$. The operation

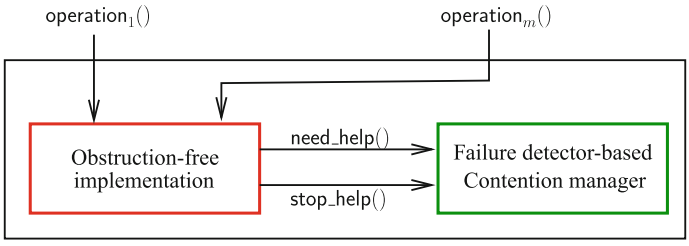

Fig. 5.13 Boosting obstruction-freedom

```
operation get_timestamp(i) is
(1)  k ← NEXT;
(2)  repeat forever
(3)    LAST[k] ← i;
(4)    if (¬COMP[k])
(5)      then COMP[k] ← true;
(6)           if (LAST[k] = i) then NEXT ← NEXT + 1; CM.stop_help(i); return(k) end if
(7)    end if;
(8)  k ← k + 1; CM.need_help(i)
(9)  end repeat
end operation.
```

Fig. 5.14 A contention-based enrichment of an obstruction-free implementation (code for p_i)

stop_help() is invoked by a process when it has obtained a timestamp as it no longer needs help.

Let us observe that the index of the invoking process is passed as a parameter when a contention manager operation is invoked. This is because progress is on processes and, consequently, a contention manager needs to know which processes have to be helped.

The next two sections present two different implementations of the contention manager object *CM*. The first makes the contention-based enriched obstruction-free implementation non-blocking (such as the one described in Fig. 5.14), while the second one makes it wait-free. It is important to notice the generic dimension of the contention manager object *CM*.

5.3.3 Boosting Obstruction-Freedom to Non-blocking

An Ω_X-based implementation of a contention manager that boosts any object implementation from obstruction-freedom to non-blocking is described in Fig. 5.15. This implementation relies on an array of SWMR atomic Boolean read/write registers *NEED_HELP*[1..*n*] with one entry per process. This array is initialized to [*false*, ..., *false*].

This contention manager compels the processes that require help to obey a simple rule: only one of them at a time is allowed to make progress. Observance of this rule is implemented thanks to the underlying failure detector Ω_X. As already indicated, each process p_i manages a local variable x containing its local view of X, which here is the current set of processes that have required help from the contention manager.

When a process p_i invokes *CM*.need_help(i), it first sets *NEED_HELP*[i] to *true*. Then, it repeatedly computes the set x of processes that have required help from the contention manager until it becomes the leader of this set. When this occurs, it returns from *CM*.need_help(i). Let us observe that the set x computed by p_i possibly changes with time. Moreover, a process may crash after it has required help. A process p_i

```
operation CM.need_help(i) is
    NEED_HELP[i] ← true;
    repeat x ← {j | NEED_HELP[j]} until (ev_leader(x) = i) end repeat;
    return()
end operation.

operation CM.stop_help(i) is NEED_HELP[i] ← false; return() end operation.
```

Fig. 5.15 A contention manager to boost obstruction-freedom to non-blocking

indicates that it no longer needs help by invoking CM.stop_help(i). Let us observe that this implementation is bounded (the array needs only n bits).

Let an *enriched* obstruction-free implementation of an object be an obstruction-free implementation of that object that invoked the operations need_help() and stop_help() of a contention manager CM (as described in Fig. 5.14, Sect. 5.3.2).

Theorem 19 *The contention manager described in Fig. 5.15 transforms an enriched obstruction-free implementation of an object into a non-blocking implementation.*

Proof Given an enriched obstruction-free implementation of an object that uses the contention manager of Fig. 5.15, let us assume (by contradiction) that this implementation is not non-blocking.

There is consequently an execution in which there is a time τ after several operations are invoked concurrently and none of them terminates. Let Q be the set of all the correct processes involved in these invocations.

Due to the enrichment of the object operations, it follows that eventually the register $NEED\_HELP[i]$ associated with each process p_i of Q remains permanently equal to *true*. Moreover, as a crashed process does not recover, there is a finite time after which the array $NEED\_HELP[1..n]$ is no longer modified. It follows that there is a time $\tau' \geq \tau$ after which all the processes of Q compute the same set $x = \{j \mid NEED\_HELP[j]\}$. Let us notice that we do not necessarily have $Q = x$, (this is due to the processes p_j that have crashed while $NEED\_HELP[j]$ is true), but we have $Q \subseteq x$.

It now follows from the validity and eventual leadership property of the failure detector instance Ω_x that there is a time $\tau'' \geq \tau'$ after which all the processes of Q have permanently the same index in their local variables ev_leader_x and this index belongs to Q. It follows from the text of CM.need_help() that this process is the only process of Q that is allowed to progress. Moreover, due to the obstruction-freedom property of the base implementation, this process then terminates its operation, which contradicts our initial assumption and concludes the proof. □

5.3.4 Boosting Obstruction-Freedom to Wait-Freedom

A $\Diamond P$-based implementation of a contention manager that boosts any object implementation from obstruction-freedom to wait-freedom is described in Fig. 5.16. Let

us remember that $\Diamond P$ provides each process p_i with a set *suspected* that eventually contains all crashed processes and only them.

This contention manager uses an underlying operation, denoted weak_ts(), that generates locally increasing timestamps such that, if a process obtains a timestamp value *ts*, then any process can obtain only a finite number of timestamp values lower than *ts*. This operation weak_ts() can be implemented from atomic read/write registers only. (Let us remark that weak_ts() is a weaker operation than the operation get_timestamp() described in Fig. 5.5.)

The internal representation of the contention manager consists of an array of SWMR atomic read/write registers $TS[1..n]$ such that only p_i can write $TS[i]$. This array is initialized to $[0, \ldots, 0]$.

When p_i invokes need_help(i), it assigns a weak timestamp to $TS[i]$ (line 1). It will reset $TS[i]$ to 0 only when it executes stop_help(i). Hence, $TS[i] \neq 0$ means that p_i is competing inside the contention manager. After it has assigned a value to $TS[i]$, p_i waits (loops) until the pair $(TS[i], i)$ is the smallest pair (according to lexicographical ordering) among the processes that (a) are competing inside the contention manager and (b) are not locally suspected to have crashed (lines 2–4).

Theorem 20 *The contention manager described in Fig. 5.16 transforms an enriched obstruction-free implementation of an object into a wait-free implementation.*

Proof The proof is similar to the proof of Theorem 19. Let us suppose (by contradiction) that there is an operation invocation by a correct process p_i that never terminates. Let ts_i be its timestamp (obtained at line 1). Moreover, let this invocation be the one with the smallest pair $\langle ts_i, i\rangle$ among all the invocations issued by correct processes that never terminate.

It follows from the property of weak_ts() that any other process obtains a finite number of timestamp values smaller than *ts*, from which we conclude that there is a finite number of operation invocations that are lexicographically ordered before $\langle ts_i, i\rangle$. Let I be this set of invocations. There are two cases.

- If an invocation of I issued by a process p_j that is not correct (i.e., a process that will crash in the execution) does not terminate, it follows from the eventual accuracy of $\Diamond P$ that eventually j is forever suspected p_i (i.e., remains forever in its set *suspected*).

```
operation need_help(i) is
(1)  if (TS[i] = 0) then TS[i] ← weak_ts() end if;
(2)  repeat competing ← {j | TS[j] ≠ 0) ∧ j ∉ suspected};
(3)        let ⟨ts, j⟩ be the smallest pair ∈ {⟨TS[x], x⟩ | x ∈ competing}
(4)  until (j = i) end repeat
end operation.

operation stop_help(i) is TS[i] ← 0; return() end operation.
```

Fig. 5.16 A contention manager to boost obstruction-freedom to wait-freedom

It then follows from the predicate tested by p_i at line 1 that there is a finite time after which, whatever the value of the pair $\langle ts_j, j\rangle$ attached to the invocation issued by p_j, j will never belong to the set *competing* repeatedly computed by p_i. Hence, these invocations cannot prevent p_i from progressing.

- Let us now consider the invocations in I issued by correct processes. Due to the definition of the pair $\langle ts_i, i\rangle$ and p_i, all these invocation terminate. Moreover, due to the definition of I, any of these processes p_j that invokes again an operation obtains a pair such that the pair $\langle ts_j, j\rangle$ is greater than the pair $\langle ts_i, i\rangle$. Consequently, the fact that j belongs or not to the set *suspected* of p_i cannot prevent p_i from progressing.

To conclude the proof, as p_i is correct, it follows from the eventual completeness property of $\Diamond P$ that there is a finite time after which i never belongs to the set $supected_k$ of any correct process p_k.

Hence, there is a finite time after which, at any correct process p_j, $i \notin suspected$ and $\langle ts_i, j\rangle$ is the smallest pair. As the number of processes is bounded, it follows that, when this occurs, only p_i can progress. □

On the design principles of contention managers As one can sec, this contention manager and the previous one are based on the same design principle. When a process asks for help, a priority is given to some process so that it can proceed alone and benefit from the obstruction-freedom property.

In the case of non-blocking, it is required that any one among the concurrent processes progresses. This was obtained from Ω_X, and the only additional underlying objects which are required are bounded atomic read/write registers. As any invocation by a correct process has to terminate, the case of wait-freedom is more demanding. This progress property is obtained from $\Diamond P$ and unbounded atomic read/write registers.

5.3.5 Mutex-Freedom Versus Loops Inside a Contention Manager Operation

In both previous contention managers, the operation need_help() contains a loop that may prevent the invoking process from making progress. But this delay period is always momentary and can never last forever.

Said differently, this loop does not simulate an implicit lock. This is due to the "eventual leadership" or "eventual accuracy" property of the underlying failure detector. Each of these "eventual" properties ensures that the processes that crash are eventually eliminated from the predicate that controls the termination of the **repeat** loop of the need_help() operation.

5.4 Summary

This chapter has introduced the notion of a mutex-free implementation and the associated progress conditions, namely obstruction-freedom, non-blocking, and wait-freedom.

To illustrate these notions, several mutex-free implementations of concurrent objects have been described: wait-free splitter, obstruction-free counter, non-blocking queue and stack based on compare&swap registers, and wait-free queues based on fetch&add registers and swap registers. Techniques based on failure detectors have also been described that allow boosting of an obstruction-free implementation of a concurrent object to a non-blocking or wait-free implementation of that object.

5.5 Bibliographic Notes

- The notion of wait-free implementation of an object is due to M. Herlihy [138].
- The notion of obstruction-free implementation is due to M. Herlihy, V. Luchangco, and M. Moir [143].
  A lot of obstruction-free, non-blocking, or wait-free implementations of queues, stacks, and other objects were developed, e.g., in [135, 142, 182, 207, 210, 238, 266, 267].

  The notion of obstruction-freedom, non-blocking, and wait-freedom are also analyzed and investigated in the following books [146, 262].
- The splitter-based obstruction-free implementation of a timestamping object described in Fig. 5.5 is from [125].

  The splitter object was informally introduced by L. Lamport in his fast mutual exclusion algorithm [191], and given an "object" status by M. Moir and J. Anderson in [209].
- The non-blocking queue based on compare&swap atomic registers is due to M. Michael and M. Scott [205].
- The non-blocking stack based on compare&swap atomic registers is due to N. Shafiei [253]. This paper presents also a proof of the stack algorithm and an implementation of a non-blocking queue based on the same principles.
- The wait-free implementation of a stack presented in Sect. 5.2.6 based on a fetch&add register and an unbounded array of swap registers is due to Y. Afek, E. Gafni, and A. Morrison [5]. A formal definition of the linearization points of the invocations of the push() and push() operations can be found in that paper.
- A methodology for creating fast wait-free data structures is described in [179]. An efficient implementation of a binary search tree is presented in [81].

- The use of failure detectors to boost obstruction-free object implementations to obtain non-blocking or wait-free implementations is due to R. Guerraoui, M. Kapałka, and P. Kuznetsov [125]. The authors show in that paper that Ω_X and $\Diamond P$ are the weakest failure detectors to boost obstruction-freedom to non-blocking and wait-freedom, respectively. "Weakest" means here that the information on failures given by each of these failure detectors is both necessary and sufficient for boosting to obstruction-freedom and wait-freedom, respectively, when one is interested in implementations based on read/write registers only.
- Ω_X was simultaneously introduced in [125, 242]. $\Diamond P$ was introduced in [67]. The concept of failure detectors is due to T.D. Chandra and S. Toueg [67]. An introduction to failure detectors can be found in [235].
- The reader interested in progress conditions can consult [161, 164, 264], which investigate the space of progress conditions from a computability point of view, and [147] which analyzes progress conditions from a dependent/independent scheduler's point of view.

5.6 Exercises and Problems

1. Prove that the concurrent queue implemented by Michael & Scott's non-blocking algorithm presented in Sect. 5.2.4 is an atomic object (i.e., its operations are atomic).

   Solution in [205].

2. The hardware-provided primitives LL(), SC() and VL() are defined in Sect. 6.3.2.

   Modify Michael & Scott's non-blocking algorithm to obtain an algorithm that uses the operations LL(), SC(), and VL() instead of compare&swap().

3. A one-shot atomic test&set register R allows each process to invoke the operation R.test&set() once. This operation is such that one of the invoking processes obtains the value *winner* while the other invoking processes obtain the value *loser*.

   Let us consider an atomic swap() operation that can be used by two (statically determined) processes only. Assuming that there are n processes, this means that there is a half-matrix of registers $MSWAP$ such that (a) $MSWAP[i, j]$ and $MSWAP[j, i]$ denote the same atomic register, (b) this register can be accessed only by p_i and p_j, and (c) their accesses are invocations of $MSWAP[j, i]$.swap().

   Design, in such a context, a wait-free algorithm that implements R.test&set().

   Solutions in [13].

4. A double-compare/single-swap operation is denoted DC&SS().

   It is a generalization of the compare&swap() operation which accesses atomically two registers at the same time. It takes three values as parameters, and its effect can be described as follows, where X and Y are the two atomic registers operated on.

   operation (X, Y).DC&SS($old1, old2, new1$):
   $prev \leftarrow X$;
   if $(X = old1 \wedge Y = old2)$ **then** $X \leftarrow new1$ **end if**;
   return($prev$).

   Design an algorithm implementing DC&SS() in a shared memory system that provides the processes with atomic registers that can be accessed with read and compare&swap() operations.

   Solutions in [135]. (The interested reader will find more general constructions of synchronization operations that atomically read and modify up to k registers in [30, 37].)

5. Define the linearization points associated with the invocations of the push() and pop() operations of the wait-free implementation of a stack presented in Sect. 5.2.6. Prove then that this implementation is linearizable (i.e., the sequence of operation invocations defined by these linearization points belongs to the sequential specification of a stack).

   Solution in [5].

6. Design a linearizable implementation of a queue (which can be accessed by any number of processes) based on fetch&add and swap registers (or on fetch&add and test&set registers). The invocations of enq() are required to be wait-free. Each invocation deq() has to return a value (i.e., it has to loop when the queue is empty). Hence, an invocation may not terminate if the queue remains empty. It is also allowed not to terminate when always overtaken by other invocations of deq().

   Solution in [148]. (Such an implementation is close to the wait-free implementation of a stack described in Sect. 5.2.6.)