Concurrent Programming: Algorithms, Principles, and Foundations

Michel Raynal

Concurrent Programming: Algorithms, Principles, and Foundations

Springer

Michel Raynal
Institut Universitaire de France
IRISA-ISTIC
Université de Rennes 1
Rennes Cedex
France

ISBN 978-3-642-32026-2 ISBN 978-3-642-32027-9 (eBook)
DOI 10.1007/978-3-642-32027-9
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012944394

ACM Computing Classification (1998): F.1, D.1, B.3

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

As long as the grass grows and the rivers flow….
From American Indians

Homo sum: humani nihil a me alienum puto.
In *Heautontimoroumenos*, Publius Terencius (194–129 BC)

… Ce jour-là j'ai bien cru tenir quelque chose et que ma vie s'en trouverait changée.
Mais rien de cette nature n'est définitivement acquis.
Comme une eau, le monde vous traverse et pour un temps vous prête ses couleurs.
Puis se retire et vous replace devant ce vide qu'on porte en soi, devant cette espèce d'insuffisance centrale de l'âme qu'il faut bien apprendre à côtoyer, à combattre, et qui, paradoxalement, est peut-être notre moteur le plus sûr.
In *L'usage du monde* (*1963*), Nicolas Bouvier (1929–1998)

What synchronization is

A concurrent program is a program made up of several entities (processes, peers, sensors, nodes, etc.) that cooperate to a common goal. This cooperation is made possible thanks to objects shared by the entities. These objects are called *concurrent objects*. Let us observe that a concurrent object can be seen as abstracting a service shared by clients (namely, the cooperating entities).

A fundamental issue of computing science and computing engineering consists in the design and the implementation of concurrent objects. In order that concurrent objects remain always consistent, the entities have to synchronize their accesses to these objects. Ensuring correct synchronization among a set of cooperating entities is far from being a trivial task. We are no longer in the world of sequential programming, and the approaches and methods used in sequential computing are of little help when one has to design concurrent programs. Concurrent programming requires not only great care but also knowledge of its scientific foundations. Moreover, concurrent programming becomes particularly difficult when one has to cope with failures of cooperating entities or concurrent objects.

Why this book?

Since the early work of E.W. Dijkstra (1965), who introduced the mutual exclusion problem, the concept of a process, the semaphore object, the notion of a weakest precondition, and guarded commands (among many other contributions), synchronization is no longer a catalog of tricks but a domain of computing science with its own concepts, mechanisms, and techniques whose results can be applied in many domains. This means that process synchronization has to be a major topic of any computer science curriculum.

This book is on synchronization and the implementation of concurrent objects. It presents in a uniform and comprehensive way the major results that have been produced and investigated in the past 30 years and have proved to be useful from both theoretical and practical points of view. The book has been written first for people who are not familiar with the topic and the concepts that are presented. These include mainly:

- Senior-level undergraduate students and graduate students in computer science or computer engineering, and graduate students in mathematics who are interested in the foundations of process synchronization.
- Practitioners and engineers who want to be aware of the state-of-the-art concepts, basic principles, mechanisms, and techniques encountered in concurrent programming and in the design of concurrent objects suited to shared memory systems.

Prerequisites for this book include undergraduate courses on algorithms and base knowledge on operating systems. Selections of chapters for undergraduate and graduate courses are suggested in the section titled “How to Use This Book” in the Afterword.

Content

As stressed by its title, this book is on algorithms, base principles, and foundations of concurrent objects and synchronization in shared memory systems, i.e., systems where the entities communicate by reading and writing a common memory. (Such a corpus of knowledge is becoming more and more important with the advent of new technologies such as multicore architectures.)

The book is composed of six parts. Three parts are more focused on base synchronization mechanisms and the construction of concurrent objects, while the other three parts are more focused on the foundations of synchronization. (A noteworthy feature of the book is that nearly all the algorithms that are presented are proved.)

- Part I is on lock-based synchronization, i.e., on well-known synchronization concepts, techniques, and mechanisms. It defines the most important synchronization problem in reliable asynchronous systems, namely the *mutual exclusion* problem (Chap. 1). It then presents several base approaches which have been proposed to solve it with machine-level instructions (Chap. 2). It also presents traditional approaches which have been proposed at a higher abstraction level to solve synchronization problems and implement concurrent objects, namely the concept of a semaphore and, at an even more abstract level, the concepts of monitor and path expression (Chap. 3).

- After the reader has become familiar with base concepts and mechanisms suited to classical synchronization in reliable systems, Part II, which is made up of a single chapter, addresses a fundamental concept of synchronization; namely, it presents and investigates the concept of *atomicity* and its properties. This allows for the formalization of the notion of a correct execution of a concurrent program in which processes cooperate by accessing shared objects (Chap. 4).

- Part I has implicitly assumed that the cooperating processes do not fail. Hence, the question: What does happen when cooperating entities fail? This is the main issue addressed in Part III (and all the rest of the book); namely, it considers that cooperating entities can halt prematurely (crash failure). To face the *net effect of asynchrony and failures*, it introduces the notions of *mutex-freedom* and associated progress conditions such as obstruction-freedom, non-blocking, and wait-freedom (Chap. 5).

The rest of Part III focuses on hybrid concurrent objects (Chap. 6), wait-free implementations of paradigmatic concurrent objects such as counters and store-collect objects (Chap. 7), snapshot objects (Chap. 8), and renaming objects (Chap. 9).

- Part IV, which is made up of a single chapter, is on *software transactional memory* systems. This is a relatively new approach whose aim is to simplify the job of programmers of concurrent applications. The idea is that programmers have to focus their efforts on which parts of their multiprocess programs have to be executed atomically and not on the way atomicity has to be realized (Chap. 10).

- Part V returns to the foundations side. It shows how reliable atomic read/write registers (shared variables) can be built from non-atomic bits. This part consists of three chapters. Chapter 11 introduces the notions of *safe* register, *regular* register, and *atomic* register. Then, Chap. 12 shows how to build an atomic bit from a safe bit. Finally, Chap. 13 shows how an atomic register of any size can be built from safe and atomic bits.

  This part shows that, while atomic read/write registers are easier to use than safe read/write registers, they are not more powerful from a computability point-of-view.

- Part VI, which concerns also the foundations side, is on the *computational power of concurrent objects*. It is made up of four chapters. It first introduces the notion of a *consensus object* and shows that consensus objects are universal objects (Chap. 14). This means that, as soon as a system provides us with atomic read/write registers and consensus objects, it is possible to implement in a wait-free manner any object defined from a sequential specification.

  Part VI then introduces the notion of *self-implementation* and shows how atomic registers and consensus objects can be built from base objects of the same type which are not reliable (Chap. 15). Then, it presents the notion of a *consensus number* and the associated *consensus hierarchy* which allows the computability power of concurrent objects to be ranked (Chap. 16). Finally, the last chapter of the book focuses on the wait-free implementation of consensus objects from read/write registers and failure detectors (Chap. 17).

To have a more complete feeling of the spirit of this book, the reader can also consult the section "What Was the Aim of This Book" in the Afterword) which describes what it is hoped has been learned from this book. Each chapter starts with a short presentation of its content and a list of keywords; it terminates with a summary of the main points that have explained and developed. Each of the six parts of the book is also introduced by a brief description of its aim and its technical content.

Acknowledgments

This book originates from lecture notes for undergraduate and graduate courses on process synchronization that I give at the University of Rennes (France) and, as an invited professor, at several universities all over the world. I would like to thank the students for their questions that, in one way or another, have contributed to this book.

I want to thank my colleagues Armando Castañeda (UNAM, MX), Ajoy Datta (UNLV, Nevada), Achour Mostéfaoui (Université de Nantes), and François Taiani (Lancaster University, UK) for their careful reading of chapters of this book. Thanks also to François Bonnet (JAIST, Kanazawa), Eli Gafni (UCLA), Damien Imbs (IRISA, Rennes), Segio Rajsbaum (UNAM, MX), Matthieu Roy (LAAS, Toulouse), and Corentin Travers (LABRI, Bordeaux) for long discussions on wait-freedom. Special thanks are due to Rachid Guerraoui (EPFL), with whom I discussed numerous topics presented in this book (and many other topics) during the past seven years. I would also like to thank Ph. Louarn (IRISA, Rennes) who was my Latex man when writing this book, and Ronan Nugent (Springer) for his support and his help in putting it all together.

Last but not least (and maybe most importantly), I also want to thank all the researchers whose results are presented in this book. Without their work, this book would not exist (Since I typeset the entire text myself (–LaTeX2$_\epsilon$ for the text and *xfig* for figures–), any typesetting or technical errors that remain are my responsibility.)

Michel Raynal
Professeur des Universités
Institut Universitaire de France
IRISA-ISTIC, Université de Rennes 1
Campus de Beaulieu, 35042 Rennes, France

September–November 2005 and June–October 2011
Rennes, Mont-Louis (ARCHI'11), Gdańsk (SIROCCO'11), Saint-Philibert,
Hong Kong (PolyU), Macau, Roma (DISC'11), Tirana (NBIS'11),
Grenoble (SSS'11), Saint-Grégoire, Douelle, Mexico City (UNAM).

Contents

Notation

No-op	No operation
Process	Program in action
n	Number of processes
Correct process	Process that does not crash during an execution
Faulty process	Process that crashes during an execution
Concurrent object	Object shared by several processes
$AA[1..m]$	Array with m entries
$\langle a, b \rangle$	Pair with two elements a and b
Mutex	Mutual exclusion
Read/write register	Synonym of read/write variable
SWSR	Single-writer/single-reader (register)
SWMR	Single-writer/multi-reader (register)
MWSR	Multi-writer/single-reader (register)
SWMR	Single-writer/multi-reader (register)
ABCD	Identifiers in italics upper case letters: shared objects
abcd	Identifiers in italics lower case letters: local variables
$\uparrow X$	Pointer to object X
$P \downarrow$	Object pointed to by the pointer P
$AA[1..s]$, $(a[1..s])$	Shared (local) array of size s
for each $i \in \{1, ..., m\}$ **do** statements **end for**	Order irrelevant
for each i **from** 1 **to** m **do** statements **end for**	Order relevant
wait (P)	**while** $\neg P$ **do** no-op **end while**
return (v)	Returns v and terminates the operation invocation
% blablabla %	Comments
;	Sequentiality operator between two statements

Figures and Algorithms

Part I
Lock-Based Synchronization

This first part of the book is devoted to lock-based synchronization, which is known as the mutual exclusion problem. It consists of three chapters:

- The first chapter is a general introduction to the mutual exclusion problem including the definition of the safety and liveness properties, which are the properties that any algorithm solving the problem has to satisfy.
- The second chapter presents three families of algorithms that solve the mutual exclusion problem. The first family is based on atomic read/write registers only, the second family is based on specific atomic hardware operations, while the third family is based on read/write registers which are weaker than atomic read/write registers.
- The last chapter of this part is on the construction of concurrent objects. Three approaches are presented. The first considers semaphores, which are traditional lock mechanisms provided at the system level. The two other approaches consider a higher abstraction level, namely the language constructs of the concept of a monitor (imperative construct) and the concept of a path expression (declarative construct).

Chapter 1
The Mutual Exclusion Problem

This chapter introduces definitions related to process synchronization and focuses then on the mutual exclusion problem, which is one of the most important synchronization problems. It also defines progress conditions associated with mutual exclusion, namely deadlock-freedom and starvation-freedom.

Keywords Competition · Concurrent object · Cooperation · Deadlock-freedom · Invariant · Liveness · Lock object · Multiprocess program · Mutual exclusion · Safety · Sequential process · Starvation-freedom · Synchronization

1.1 Multiprocess Program

1.1.1 The Concept of a Sequential Process

A *sequential algorithm* is a formal description of the behavior of a sequential state machine: the text of the algorithm states the transitions that have to be sequentially executed. When written in a specific programming language, an algorithm is called a *program*.

The concept of a *process* was introduced to highlight the difference between an algorithm as a text and its execution on a processor. While an algorithm is a text that describes statements that have to be executed (such a text can also be analyzed, translated, etc.), a process is a "text in action", namely the dynamic entity generated by the execution of an algorithm (program) on one or several processors. At any time, a process is characterized by its state (which comprises, among other things, the current value of its program counter). A sequential process (sometimes called a *thread*) is a process defined by a single control flow (i.e., its behavior is managed by a single program counter).

1.1.2 The Concept of a Multiprocess Program

The concept of a process to express the idea of an activity has become an indispensable tool to master the activity on multiprocessors. More precisely, a concurrent algorithm (or concurrent program) is the description of a set of sequential state machines that cooperate through a communication medium, e.g., a shared memory. A concurrent algorithm is sometimes called a multiprocess program (each process corresponding to the sequential execution of a given state machine).

This chapter considers processes that are reliable and asynchronous. "*Reliable*" means that each process results from the correct execution of the code of the corresponding algorithm. "*Asynchronous*" means that there is no timing assumption on the time it takes for a process to proceed from a state transition to the next one (which means that an asynchronous sequential process proceeds at an arbitrary speed).

1.2 Process Synchronization

1.2.1 Processors and Processes

Processes of a multiprocess program *interact* in one way or another (otherwise, each process would be independent of the other processes, and a set of independent processes does not define a multiprocess program). Hence, the processes of a multiprocess program do interact and may execute simultaneously (we also say that the processes execute "in parallel" or are "concurrent").

In the following we consider that there is one processor per process and consequently the processes do execute in parallel. This assumption on the number of processors means that, when there are fewer processors than processes, there is an underlying *scheduler* (hidden to the processes) that assigns processors to processes. This scheduling is assumed to be fair in the sense that each process is repeatedly allowed a processor for finite periods of time. As we can see, this is in agreement with the asynchrony assumption associated with the processes because, when a process is waiting for a processor, it does not progress, and consequently, there is an arbitrary period of time that elapses between the last state transition it executed before stopping and the next state transition that it will execute when again assigned a processor.

1.2.2 Synchronization

Process synchronization occurs when the progress of one or several processes depends on the behavior of other processes. Two types of process interaction require synchronization: competition and cooperation.

More generally, *synchronization* is the set of rules and mechanisms that allows the specification and implementation of sequencing properties on statements issued by the processes so that all the executions of a multiprocess program are correct.

1.2.3 Synchronization: Competition

This type of process interaction occurs when processes have to compete to execute some statements and only one process at a time (or a bounded number of them) is allowed to execute them. This occurs, for example, when processes compete for a shared resource. More generally, resource allocation is a typical example of process competition.

A simple example As an example let us consider a random access input/output device such as a shared disk. Such a disk provides the processes with three primitives: seek(x), which moves the disk read/write head to the address x; read(), which returns the value located at the current position of the read/write head; and write(v), which writes value v at the current position of the read/write head.

Hence, if a process wants to read the value at address x of a disk D, it has to execute the operation disk_read(x) described in Fig. 1.1. Similarly, if a process wants to write a new value v at address x, it has to execute the operation disk_write(x, v) described in the same figure.

The disk primitives seek(), read(), and write() are implemented in hardware, and each invocation of any of these primitives appears to an *external observer* as if it was executed instantaneously at a single point of the time line between the beginning and the end of its real-time execution. Moreover, no two primitive invocations are associated with the same point of the time line. Hence, the invocations appear as if they had been executed sequentially. (This is the *atomicity consistency condition* that will be more deeply addressed in Chap. 4.)

If a process p invokes disk_read(x) and later (after p's invocation has terminated) another process q invokes disk_write(y, v), everything works fine (both operations execute correctly). More precisely, the primitives invoked by p and q have been invoked sequentially, with first the invocations by p followed by the invocations by q;

```
operation disk_read($x$) is
    % $r$ is a local variable of the invoking process %
    $D$.seek($x$); $r \leftarrow D$.read(); return($r$)
end operation.

operation disk_write($x, v$) is
    $D$.seek($x$); $D$.write($v$); return()
end operation.
```

Fig. 1.1 Operations to access a disk

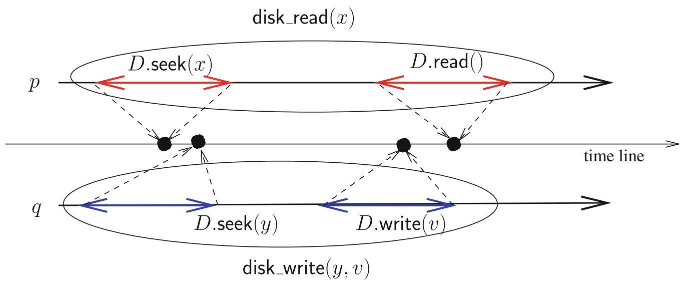


Fig. 1.2 An interleaving of invocations to disk primitives

i.e., from the disk D point of view, the execution corresponds to the sequence D.seek(x); $r \leftarrow D$.read(); D.seek(y); D.write(v), from which we conclude that p has read the value at address x and afterwards q has written the value v at address y.

Let us now consider the case where p and q simultaneously invoke disk_read(x) and disk_write(y, v), respectively. The effect of the corresponding parallel execution is produced by any interleaving of the primitives invoked by p and the primitives invoked by q that respects the order of invocations issued by p and q. As an example, a possible execution is depicted in Fig. 1.2. This figure is a classical space-time diagram. Time flows from left to right, and each operation issued by a process is represented by a segment on the time axis associated with this process. Two dashed arrows are associated with each invocation of an operation. They meet at a point of the "real time" line, which indicates the instant at which the corresponding operation appears to have been executed instantaneously. This sequence of points define the order in which the execution is seen by an external sequential observer (i.e., an observer who can see one operation invocation at a time).

In this example, the processes p and q have invoked in parallel D.seek(x) and D.seek(y), respectively, and D.seek(x) appears to be executed before D.seek(y). Then q executes D.write(v) while p executes in parallel D.read(), and the write by q appears to an external observer to be executed before the read of p.

It is easy to see that, while the write by process q is correct (namely v has been written at address y), the read by process p of the value at address x is incorrect (p obtains the value written at address y and not the value stored at address x). Other incorrect parallel executions (involving invocations of both disk_read() and disk_write() or involving only invocations of disk_write() operations) in which a value is not written at the correct address can easily be designed.

A solution to prevent this problem from occurring consists in allowing only one operation at a time (either disk_read() or disk_write()) to be executed. Mutual exclusion (addressed later in this chapter) provides such a solution.

Non-determinism It is important to see that parallelism (or concurrency) generates non-determinism: the interleaving of the invocations of the primitives cannot be predetermined, it depends on the execution. Preventing interleavings that would produce incorrect executions is one of the main issues of synchronization.

1.2.4 Synchronization: Cooperation

This section presents two examples of process cooperation. The first is a pure coordination problem while the second is the well-known producer–consumer problem. In both cases the progress of a process may depend on the progress of other processes.

Barrier (or rendezvous) A synchronization barrier (or rendezvous) is a set of control points, one per process involved in the barrier, such that each process is allowed to pass its control point only when all other processes have attained their control points.

From an operational point of view, each process has to stop until all other processes have arrived at their control point. Differently from mutual exclusion (see below), a barrier is an instance of the *mutual coincidence* problem.

A producer–consumer problem Let us consider two processes, one called "the producer" and the other called "the consumer", such that the producer produces data items that the consumer consumes (this cooperation pattern, called producer–consumer, occurs in a lot of applications). Assuming that the producer loops forever on producing data items and the consumer loops forever on consuming data items, the problem consists in ensuring that (a) only data items that were produced are consumed, and (b) each data item that was produced is consumed exactly once.

One way to solve this problem could be to use a synchronization barrier: Both the producer (when it has produced a new data item) and the consumer (when it wants to consume a new data item) invoke the barrier operation. When, they have both attained their control point, the producer gives the data item it has just produced to the consumer. This coordination pattern works but is not very efficient (overly synchronized): for each data item, the first process that arrives at its control point has to wait for the other process.

An easy way to cope with this drawback and increase concurrency consists in using a shared buffer of size $k \geq 1$. Such an object can be seen as queue or a circular array. When it has produced a new data item, the producer adds it to the end of the queue. When it wants to consume a new item, the consumer process withdraws the data item at the head of the queue. With such a buffer of size k, a producer has to wait only when the buffer is full (it then contains k data items produced and not yet consumed). Similarly, the consumer has to wait only when the buffer is empty (which occurs each time all data items that have been produced have been consumed).

1.2.5 The Aim of Synchronization Is to Preserve Invariants

To better understand the nature of what synchronization is, let us consider the previous producer–consumer problem. Let $\#p$ and $\#c$ denote the number of data items produced and consumed so far, respectively. The instance of the problem

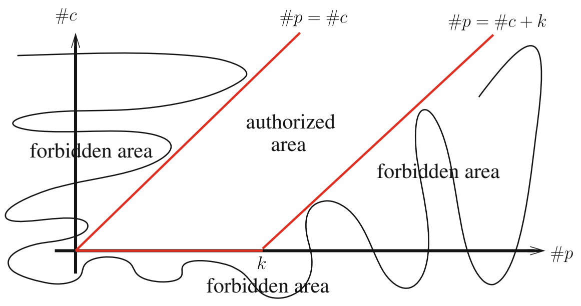


Fig. 1.3 Synchronization is to preserve invariants

associated with a buffer of size k is characterized by the following invariant: $(\#c \geq 0) \wedge (\#p \geq \#c) \wedge (\#p \leq \#c + k)$. The predicate $\#c \geq 0$ is trivial. The predicate $\#p \geq \#c$ states that the number of data items that have been consumed cannot be greater than the number of data items that have been produced, while the predicate $\#p \leq \#c + k$ states that the size of the buffer is k.

This invariant is depicted in Fig. 1.3, where any point $(\#p, \#c)$ inside the area (including its borders) defined by the lines $\#c = 0$, $\#p = \#c$, and $\#p = \#c + k$ is a correct pair of values for $\#p$ and $\#c$. This means that, in order to be correct, the synchronization imposed to the processes must ensure that, in any execution and at any time, the current pair $(\#p, \#c)$ has to remain inside the authorized area. This shows that the aim of synchronization is to preserve invariants. More precisely, when an invariant is about to be violated by a process, that process has to be stopped until the values of the relevant state variables allow it to proceed: to keep the predicate $\#p \leq \#c + k$ always satisfied, the producer can produce only when $\#p < \#c + k$; similarly, in order for the predicate $\#c \leq \#p$ to be always satisfied, the consumer can consume only when $\#c < \#p$. In that way, the pair $(\#p, \#c)$ will remain forever in the authorized area.

It is possible to represent the previous invariant in a way that relates the control flows of both the producer and the consumer. Let p^i and c^j represent the ith data item production and the jth data item consumption, respectively. Let $a \rightarrow b$ means that a has to be terminated before b starts (where each of a and b is some p^i or c^j). A control flow-based statement of the invariant $(\#c \geq 0) \wedge (\#p \geq \#c) \wedge (\#p \leq \#c + k)$ is expressed in Fig. 1.4.

$$\begin{tikzcd}
p^1 \arrow[r] \arrow[d] & p^2 \arrow[r, dash, dashed] \arrow[d] & p^i \arrow[r] \arrow[d] & p^{i+1} \arrow[r, dashed] \arrow[d] & p^{i+k} \arrow[r, dash, dashed] \arrow[d] & {} \\
c^1 \arrow[r] & c^2 \arrow[r, dash, dashed] & c^i \arrow[r] \arrow[urr] & c^{i+1} \arrow[r, dashed] & c^{i+k} \arrow[r, dash, dashed] & {}
\end{tikzcd}$$

Fig. 1.4 Invariant expressed with control flows

1.3 The Mutual Exclusion Problem

1.3.1 The Mutual Exclusion Problem (Mutex)

Critical section Let us consider a part of code A (i.e., an algorithm) or several parts of code $A, B, C \ldots$ (i.e., different algorithms) that, for some consistency reasons, must be executed by a single process at a time. This means that, if a process is executing one of these parts of code, e.g., the code B, no other process can simultaneously execute the same or another part of code, i.e., any of the codes A or B or C or etc. This is, for example, the case of the disk operations disk_read() and disk_write() introduced in Sect. 1.2.2, where guaranteeing that, at any time, at most one process can execute either of these operations ensures that each read or write of the disk is correct. Such parts of code define what is called a *critical section*. It is assumed that a code defining a critical section always terminates when executed by a single process at a time.

In the following, the critical section code is abstracted into a procedure called cs_code(in) where in denotes its input parameter (if any) and that returns a result value (without loss of generality, the default value $\perp$ is returned if there is no explicit result).

Mutual exclusion: providing application processes with an appropriate abstraction level The *mutual exclusion* problem (sometimes abbreviated *mutex*) consists in designing an *entry algorithm* (also called entry protocol) and an *exit algorithm* (also called exit protocol) that, when used to bracket a critical section cs_code(in), ensure that the critical section code is executed by at most one process at a time.

Let acquire_mutex() and release_mutex() denote these "bracket" operations. When several processes are simultaneously executing acquire_mutex(), we say that they are competing for access to the critical section code. If one of these invocations terminates while the other invocations do not, the corresponding process p is called the *winner*, while each other competing process q is a *loser* (its invocation remains pending). When considering the pair (p, q) of competing processes, we say that p has won its competition with q.

It is assumed that the code of the processes are well formed, which means that, each time a process wants to execute cs_code(), it first executes acquire_mutex(), then executes cs_code(), and finally executes release_mutex(). It is easy to direct the processes to be well-formed by providing them with a high-level procedure which encapsulates the critical section code cs_code(). This procedure, denoted protected_code(in), is defined as follows (r is a local variable of the invoking process):

```
procedure protected_code($in$) is
    acquire_mutex(); $r \leftarrow$ cs_code($in$); release_mutex(); return($r$)
end procedure.
```

Mutual exclusion: definition The mutual exclusion problem consists in implementing the operations acquire_mutex() and release_mutex() in such a way that the following properties are always satisfied:

- Mutual exclusion, i.e., at most one process at a time executes the critical section code.
- Starvation-freedom. Whatever the process p, each invocation of acquire_mutex() issued by p eventually terminates.

A problem is defined by *safety* properties and *liveness* properties. Safety properties state that nothing bad happens. They can usually be stated as invariants. This invariant is here the mutual exclusion property which states that at most one process at a time can execute the critical section code.

A solution in which no process is ever allowed to execute the critical section code would trivially satisfy the safety property. This trivial "solution" is prevented by the starvation-freedom liveness property, which states that, if a process wants to execute the critical section code, then that process eventually executes it.

On liveness properties Starvation-freedom means that a process that wants to enter the critical section can be bypassed an arbitrary but *finite* number of times by each other process. It is possible to define liveness properties which are weaker or stronger than starvation-freedom, namely deadlock-freedom and bounded bypass.

- Deadlock-freedom. Whatever the time τ, if before τ one or several processes have invoked the operation acquire_mutex() and none of them has terminated its invocation at time τ, then there is a time $\tau' > \tau$ at which a process that has invoked acquire_mutex() terminates its invocation.

Let us notice that deadlock-freedom does not require the process that terminates its invocation of acquire_mutex() to be necessarily one of the processes which have invoked acquire_mutex() before time τ. It can be a process that has invoked acquire_mutex() after time τ. The important point is that, as soon as processes want to enter the critical section, then processes will enter it.

It is easy to see that starvation-freedom implies deadlock-freedom, while deadlock-freedom does not imply starvation-freedom. This is because, if permanently several processes are concurrently executing acquire_mutex(), it is possible that some of them never win the competition (i.e., never terminate their execution of acquire_mutex()). As an example, let us consider three processes p_1, p_2, and p_3 that are concurrently executing acquire_mutex() and p_1 wins (terminates). Due to the safety property, there is a single winner at a time. Hence, p_1 executes the procedure cs_code() and then release_mutex(). Then, p_2 wins the competition with p_3 and starts executing cs_code(). During that time, p_1 invokes acquire_mutex() to execute cs_code() again. Hence, while p_3 is executing acquire_mutex(), it has lost two competitions: the first one with respect to p_1 and the second one with respect to p_2. Moreover, p_3 is currently competing again with p_1. When later p_2 terminates its execution of release_mutex(), p_1 wins the competition with p_3 and starts its second execution of cs_code(). During that time p_2 invokes acquire_mutex() again, etc.

It is easy to extend this execution in such a way that, while p_3 wants to enter the critical section, it can never enter it. This execution is deadlock-free but (due to p_3) is not starvation-free.

Service point of view versus client point of view Deadlock-freedom is a meaningful liveness condition from the critical section (service) point of view: if processes are competing for the critical section, one of them always wins, hence the critical section is used when processes want to access it. On the other hand, starvation-freedom is a meaningful liveness condition from a process (client) point of view: whatever the process p, if p wants to execute the critical section code it eventually executes it.

Finite bypass versus bounded bypass A liveness property that is stronger than starvation-freedom is the following one. Let p and q be a pair of competing processes such that q wins the competition. Let $f(n)$ denote a function of n (where n is the total number of processes).

- Bounded bypass. There is a function $f(n)$ such that, each time a process invokes acquire_mutex(), it loses at most $f(n)$ competitions with respect to the other processes.

Let us observe that starvation-freedom is nothing else than the case where the number of times that a process can be bypassed is finite. More generally, we have the following hierarchy of liveness properties: bounded bypass $\Rightarrow$ starvation-freedom $\equiv$ finite bypass $\Rightarrow$ deadlock-freedom.

1.3.2 Lock Object

Definition A lock (say *LOCK*) is a shared object that provides the processes with two operations denoted *LOCK*.acquire_lock() and *LOCK*.release_lock(). It can take two values, *free* and *locked*, and is initialized to the value *free*. Its behavior is defined by a sequential specification: from an external observer point of view, all the acquire_lock() and release_lock() invocations appear as if they have been invoked one after the other. Moreover, using the regular language operators ";" and "$*$", this sequence corresponds to the regular expression $\big(LOCK.\mathsf{acquire\_lock}();\ LOCK.\mathsf{release\_lock}()\big)^*$ (see Fig. 1.5).

Lock versus Mutex It is easy to see that, considering acquire_lock() as a synonym of acquire_mutex() and release_lock() as a synonym of release_mutex(), a lock object solves the mutual exclusion problem. Hence, the lock object is the object associated with mutual exclusion: solving the mutual exclusion problem is the same as implementing a lock object.

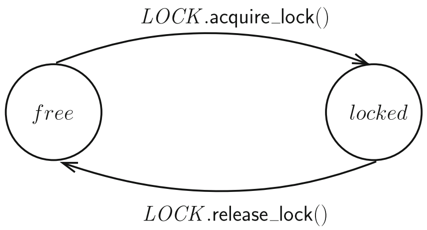


Fig. 1.5 Sequential specification of a lock object *LOCK*

1.3.3 Three Families of Solutions

According to the operations and their properties provided to the processes by the underlying shared memory communication system, several families of mutex algorithms can be designed. We distinguish three distinct families of mutex algorithms which are investigated in the next chapter.

Atomic read/write registers In this case the processes communicate by reading and writing shared atomic registers. There is no other way for them to cooperate. Atomic registers and a few mutex algorithms based on such registers are presented in Sect. 2.1.

Specialized hardware primitives Multiprocessor architectures usually offer hardware primitives suited to synchronization. These operations are more sophisticated than simple read/write registers. Some of them will be introduced and used to solve mutual exclusion in Sect. 2.2.

Mutex without underlying atomicity Solving the mutual exclusion problem allows for the construction of high-level atomic operations (i.e., whatever the base statements that define a block of code, this block of code can be made atomic). The mutex algorithms based on atomic read/write registers or specialized hardware primitives assume that the underlying shared memory offers low-level atomic operations and those are used to implement mutual exclusion at a higher abstraction level. This means that these algorithms are atomicity-based: they allow high level programmer-defined atomic operations to be built from base hardware-provided atomic operations. Hence, the fundamental question: Can programmer-defined atomic operations be built without assuming atomicity at a lower abstraction level? This question can also be stated as follows: Is atomicity at a lower level required to solve atomicity at a higher level?

Somehow surprisingly, it is shown in Sect. 2.3 that the answer to the last formulation of the previous question is "no". To that end, new types of shared read/write registers are introduced and mutual exclusion algorithms based on such particularly weak registers are presented.

1.4 Summary

This chapter has presented the mutual exclusion problem. Solving this problem consists in providing a lock object, i.e., a synchronization object that allows a zone of code to be bracketed to guarantee that a single process at a time can execute it.

1.5 Bibliographic Notes

- The mutual exclusion problem was first stated by E.W. Dijkstra [88].
- A theory of interprocess communication and mutual exclusion is described in [185].
- The notions of safety and liveness were introduced by L. Lamport in [185]. The notion of liveness is investigated in [20].
- An invariant-based view of synchronization is presented in [194].

Chapter 2
Solving Mutual Exclusion

This chapter is on the implementation of mutual exclusion locks. As announced at the end of the previous chapter, it presents three distinct families of algorithms that solve the mutual exclusion problem. The first is the family of algorithms which are based on atomic read/write registers only. The second is the family of algorithms which are based on specialized hardware operations (which are atomic and stronger than atomic read/write operations). The third is the family of algorithms which are based on read/write registers which are weaker than atomic registers. Each algorithm is first explained and then proved correct. Other properties such as time complexity and space complexity of mutual exclusion algorithms are also discussed.

Keywords Atomic read/write register · Lock object · Mutual exclusion · Safe read/write register · Specialized hardware primitive (test&set, fetch&add, compare&swap)

2.1 Mutex Based on Atomic Read/Write Registers

2.1.1 Atomic Register

The *read/write register* object is one of the most basic objects encountered in computer science. When such an object is accessed only by a single process it is said to be *local* to that process; otherwise, it is a *shared* register. A local register allows a process to store and retrieve data. A shared register allows concurrent processes to also exchange data.

Definition A *register R* can be accessed by two base operations: $R.\mathsf{read}()$, which returns the value of R (also denoted $x \leftarrow R$ where x is a local variable of the invoking process), and $R.\mathsf{write}(v)$, which writes a new value into R (also denoted $R \leftarrow v$, where v is the value to be written into R). An *atomic* shared register satisfies the following properties:

- Each invocation op of a read or write operation:
  - Appears as if it was executed at a single point $\tau(\mathsf{op})$ of the time line,
  - $\tau(\mathsf{op})$ is such that $\tau_b(\mathsf{op}) \leq \tau(\mathsf{op}) \leq \tau_e(\mathsf{op})$, where $\tau_b(\mathsf{op})$ and $\tau_e(\mathsf{op})$ denote the time at which the operation op started and finished, respectively,
  - For any two operation invocations op1 and op2: $(\mathsf{op1} \neq \mathsf{op2}) \Rightarrow \big(\tau(\mathsf{op1}) \neq \tau(\mathsf{op2})\big)$.
- Each read invocation returns the value written by the closest preceding write invocation in the sequence defined by the $\tau()$ instants associated with the operation invocations (or the initial value of the register if there is no preceding write operation).

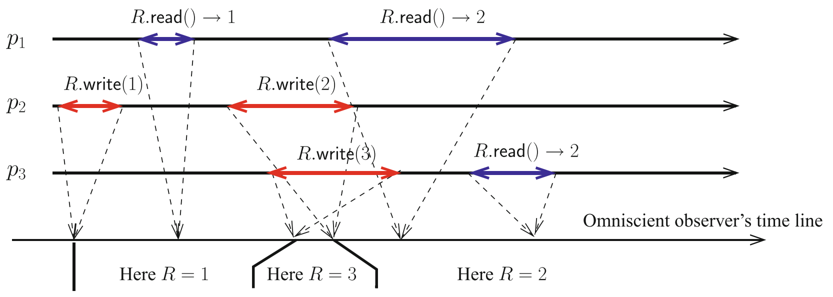
This means that an atomic register is such that all its operation invocations *appear* as if they have been executed sequentially: any invocation op1 that has terminated before an invocation op2 starts appears before op2 in that sequence, and this sequence belongs to the specification of a sequential register.

An atomic register can be single-writer/single-reader (SWSR)—the reader and the writer being distinct processes—or single-writer/multi-reader (SWMR), or multi-writer/multi-reader (MWMR) . We assume that a register is able to contain any value. (As each process is sequential, a local register can be seen as a trivial instance of an atomic SWSR register where, additionally, both the writer and the reader are the same process.)

An example An execution of a MWMR atomic register accessed by three processes p_1, p_2, and p_3 is depicted in Fig. 2.1 using a classical space-time diagram. $R.\mathsf{read}() \rightarrow v$ means that the corresponding read operation returns the value v. Consequently, an external observer sees the following sequential execution of the register R which satisfies the definition of an atomic register:

$$R.\mathsf{write}(1),\ R.\mathsf{read}() \rightarrow 1,\ R.\mathsf{write}(3),\ R.\mathsf{write}(2),\ R.\mathsf{read}() \rightarrow 2,\ R.\mathsf{read}() \rightarrow 2.$$

Let us observe that $R.\mathsf{write}(3)$ and $R.\mathsf{write}(2)$ are concurrent, which means that they could appear to an external observer as if $R.\mathsf{write}(2)$ was executed before



p_1
p_2
p_3
$R.\mathsf{read}() \rightarrow 1$
$R.\mathsf{read}() \rightarrow 2$
$R.\mathsf{write}(1)$
$R.\mathsf{write}(2)$
$R.\mathsf{write}(3)$
$R.\mathsf{read}() \rightarrow 2$
Omniscient observer's time line
Here $R = 1$
Here $R = 3$
Here $R = 2$

Fig. 2.1 An atomic register execution

$R.\mathsf{write}(3)$. If this was the case, the execution would be correct if the last two read invocations (issued by p_1 and p_3) return the value 3; i.e., the external observer should then see the following sequential execution:

$$R.\mathsf{write}(1),\ R.\mathsf{read}() \to 1,\ R.\mathsf{write}(2),\ R.\mathsf{write}(3),\ R.\mathsf{read}() \to 3,\ R.\mathsf{read}() \to 3.$$

Let us also observe that the second read invocation by p_1 is concurrent with both $R.\mathsf{write}(2)$ and $R.\mathsf{write}(3)$. This means that it could appear as having been executed before these two write operations or even between them. If it appears as having been executed before these two write operations, it should return the value 1 in order for the register behavior be atomic.

As shown by these possible scenarios (and as noticed before) *concurrency* is intimately related to *non-determinism*. It is not possible to predict which execution will be produced; it is only possible to enumerate the set of possible executions that could be produced (we can only predict that the one that is actually produced is one of them).

Examples of non-atomic read and write operations will be presented in Sect. 2.3.

Why atomicity is important Atomicity is a fundamental concept because it allows the composition of shared objects for free (i.e., their composition is at no additional cost). This means that, when considering two (or more) atomic registers $R1$ and $R2$, the composite object $[R1, R2]$ which is made up of $R1$ and $R2$ and provides the processes with the four operations $R1.\mathsf{read}()$, $R1.\mathsf{write}()$, $R2.\mathsf{read}()$, and $R2.\mathsf{write}()$ is also atomic. Everything appears as if at most one operation at a time was executed, and the sub-sequence including only the operations on $R1$ is a correct behavior of $R1$, and similarly for $R2$.

This is very important when one has to reason about a multiprocess program whose processes access atomic registers. More precisely, we can keep *reasoning sequentially* whatever the number of atomic registers involved in a concurrent computation. Atomicity allows us to reason on a set of atomic registers as if they were a single "bigger" atomic object. Hence, we can reason in terms of sequences, not only for each atomic register taken separately, but also on the whole set of registers as if they were a single atomic object.

The composition of atomic objects is formally addressed in Sect. 4.4, where it is shown that, as atomicity is a "local property", atomic objects compose for free.

2.1.2 Mutex for Two Processes: An Incremental Construction

The mutex algorithm for two processes that is presented below is due to G.L. Peterson (1981). This construction, which is fairly simple, is built from an "addition" of two base components. Despite the fact that these components are nearly trivial, they allow us to introduce simple basic principles.

```
operation acquire_mutex1$(i)$ is
    $AFTER\_YOU \leftarrow i$; wait $(AFTER\_YOU \neq i)$; return()
end operation.

operation release_mutex1$(i)$ is return() end operation.
```

Fig. 2.2 Peterson's algorithm for two processes: first component (code for p_i)

The processes are denoted p_i and p_j. As the algorithm for p_j is the same as the one for p_i after having replaced i by j, we give only the code for p_i.

First component This component is described in Fig. 2.2 for process p_i. It is based on a single atomic register denoted *AFTER_YOU*, the initial value of which is irrelevant (a process writes into this register before reading it). The principle that underlies this algorithm is a "politeness" rule used in current life. When p_i wants to acquire the critical section, it sets *AFTER_YOU* to its identity i and waits until $AFTER\_YOU \neq i$ in order to enter the critical section. Releasing the critical section entails no particular action.

It is easy to see that this algorithm satisfies the mutual exclusion property. When both processes want to acquire the critical section, each assigns its identity to the register *AFTER_YOU* and waits until this register contains the identity of the other process. As the register is atomic, there is a "last" process, say p_j, that updated it, and consequently only the other process p_i can proceed to the critical section.

Unfortunately, this simple algorithm is not deadlock-free. If one process alone wants to enter the critical section, it remains blocked forever in the **wait** statement. Actually, this algorithm ensures that, when both processes want to enter the critical section, the first process that updates the register *AFTER_YOU* is the one that is allowed to enter it.

Second component This component is described in Fig. 2.3. It is based on a simple idea. Each process p_i manages a flag (denoted $FLAG[i]$) the value of which is $down$ or up. Initially, both flags are down. When a process wants to acquire the critical section, it first raises its flag to indicate that it is interested in the critical section. It is then allowed to proceed only when the flag of the other process is equal to $down$.

To release the critical section, a process p_i has only to reset $FLAG[i]$ to its initial value (namely, $down$), thereby indicating that it is no longer interested in the mutual exclusion.

```
operation acquire_mutex2$(i)$ is
    $FLAG[i] \leftarrow up$; wait $(FLAG[j] = down)$; return()
end operation.

operation release_mutex2$(i)$ is $FLAG[i] \leftarrow down$; return() end operation.
```

Fig. 2.3 Peterson's algorithm for two processes: second component (code for p_i)

It is easy to see that, if a single process p_i wants to repeatedly acquire the critical section while the other process is not interested in the critical section, it can do so (hence this algorithm does not suffer the drawback of the previous one). Moreover, it is also easy to see that this algorithm satisfies the mutual exclusion property. This follows from the fact that each process follows the following pattern: first write its flag and only then read the value of the other flag. Hence, assuming that p_i has acquired (and not released) the critical section, we had $(FLAG[i] = up) \wedge (FLAG[j] = down)$ when it was allowed to enter the critical section. It follows that, after p_j has set $FLAG[j]$ to the value up, it reads up from $FLAG[i]$ and is delayed until p_i resets $FLAG[i]$ to $down$ when it releases the critical section.

Unfortunately, this algorithm is not deadlock-free. If both processes concurrently raise first their flags and then read the other flag, each process remains blocked until the other flag is set down which will never be done.

Remark: the notion of a livelock In order to prevent the previous deadlock situation, one could think replacing **wait** $(FLAG[j] = down)$ by the following statement:

```
while $(FLAG[j] = up)$ do
        $FLAG[i] \leftarrow down$;
        $p_i$ delays itself for an arbitrary period of time;
        $FLAG[i] \leftarrow up$
end while.
```

This modification can reduce deadlock situations but cannot eliminate all of them. This occurs, for example when both processes execute "synchronously" (both delay themselves for the same duration and execute the same step—writing their flag and reading the other flag—at the very same time). When it occurs, this situation is sometimes called a *livelock*.

This tentative solution was obtained by playing with asynchrony (modifying the process speed by adding delays). As a correct algorithm has to work despite any asynchrony pattern, playing with asynchrony can eliminate bad scenarios but cannot suppress all of them.

2.1.3 A Two-Process Algorithm

Principles and description In a very interesting way, a simple "addition" of the two previous "components" provides us with a correct mutex algorithm for two processes (Peterson's two-process algorithm). This component addition consists in a process p_i first raising its flag (to indicate that it is competing, as in Fig. 2.3), then assigning its identity to the atomic register *AFTER_YOU* (as in Fig. 2.2), and finally waiting until any of the progress predicates $AFTER\_YOU \neq i$ or $FLAG[j] = down$ is satisfied.

It is easy to see that, when a single process wants to enter the critical section, the flag of the other process allows it to enter. Moreover, when each process sees that

```
operation acquire_mutex($i$) is
    $FLAG[i] \leftarrow up$;
    $AFTER\_YOU \leftarrow i$;
    wait $\big((FLAG[j] = down) \vee (AFTER\_YOU \neq i)\big)$;
    return()
end operation.

operation release_mutex($i$) is $FLAG[i] \leftarrow down$; return() end operation.
```

Fig. 2.4 Peterson's algorithm for two processes (code for p_i)

the flag of the other one was raised, the current value of the register *AFTER_YOU* allows exactly one of them to progress.

It is important to observe that, in the **wait** statement of Fig. 2.4, the reading of the atomic registers *FLAG*[j] and *AFTER_YOU* are asynchronous (they are done at different times and can be done in any order).

Theorem 1 *The algorithm described in Fig. 2.4 satisfies mutual exclusion and bounded bypass* (*where the bound is* $f(n) = 1$).

Preliminary remark for the proof The reasoning is based on the fact that the three registers *FLAG*[i], *FLAG*[j], and *AFTER_YOU* are atomic. As we have seen when presenting the atomicity concept (Sect. 2.1.1), this allows us to reason as if at most one read or write operation on any of these registers occurs at a time.

Proof Proof of the mutual exclusion property.
Let us assume by contradiction that both p_i and p_j are inside the critical section. Hence, both have executed acquire_mutex() and we have then *FLAG*[i] $=$ *up*, *FLAG*[j] $=$ *up* and *AFTER_YOU* $= j$ (if *AFTER_YOU* $= i$, the reasoning is the same after having exchanged i and j). According to the predicate that allowed p_i to enter the critical section, there are two cases.

- Process p_i has terminated acquire_mutex(i) because *FLAG*[j] $= down$.

  As p_i has set *FLAG*[i] to *up* before reading *down* from *FLAG*[j] (and entering the critical section), it follows that p_j cannot have read *down* from *FLAG*[i] before entering the critical section (see Fig. 2.5). Hence, p_j entered it due to the predicate *AFTER_YOU* $= i$. But this contradicts the assumption that *AFTER_YOU* $= j$ when both processes are inside the critical section.

- Process p_i has terminated acquire_mutex(i) because *AFTER_YOU* $= j$.

  As (by assumption) p_j is inside the critical section, *AFTER_YOU* $= j$, and only p_j can write j into *AFTER_YOU*, it follows that p_j has terminated acquire_mutex(j) because it has read *down* from *FLAG*[i]. On another side, *FLAG*[i] remains continuously equal to *up* from the time at which p_i has executed the first statement of acquire_mutex(i) and the execution of release_mutex(i) (Fig. 2.6).

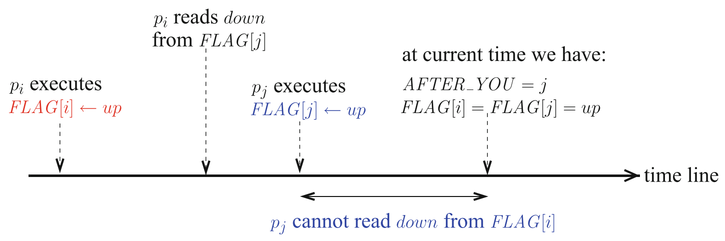

p_i reads $down$ from $FLAG[j]$
at current time we have:
p_i executes $FLAG[i] \leftarrow up$
p_j executes $FLAG[j] \leftarrow up$
$AFTER\_YOU = j$
$FLAG[i] = FLAG[j] = up$
time line
p_j cannot read $down$ from $FLAG[i]$

Fig. 2.5 Mutex property of Peterson's two-process algorithm (part 1)

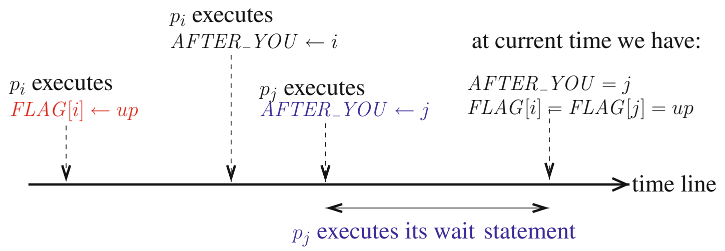

p_i executes $AFTER\_YOU \leftarrow i$
at current time we have:
p_i executes $FLAG[i] \leftarrow up$
p_j executes $AFTER\_YOU \leftarrow j$
$AFTER\_YOU = j$
$FLAG[i] = FLAG[j] = up$
time line
p_j executes its wait statement

Fig. 2.6 Mutex property of Peterson's two-process algorithm (part 2)

As p_j executes the **wait** statement after writing j into *AFTER_YOU* and p_i read j from *AFTER_YOU*, it follows that p_j cannot read $down$ from $FLAG[i]$ when it executes the **wait** statement. This contradicts the assumption that p_j is inside the critical section.

Proof of the bounded bypass property.
Let p_i be the process that invokes acquire_mutex(i). If $FLAG[j] = down$ or $AFTER\_YOU = j$ when p_i executes the **wait** statement, it enters the critical section.

Let us consequently assume that $(FLAG[j] = up) \wedge (AFTER\_YOU = i)$ when p_i executes the **wait** statement (i.e., the competition is lost by p_i). If, after p_j has executed release_mutex(j), it does not invoke acquire_mutex(j) again, we permanently have $FLAG[j] = down$ and p_i eventually enters the critical section.

Hence let us assume that p_j invokes again acquire_mutex(j) and sets $FLAG[j]$ to up before p_i reads it. Thus, the next read of $FLAG[j]$ by p_i returns up. We have then $(FLAG[j] = up) \wedge (AFTER\_YOU = i)$, and p_i cannot progress (see Fig. 2.7).

It follows from the code of acquire_mutex(j) that p_j eventually assigns j to *AFTER_YOU* (and the predicate $AFTER\_YOU = j$ remains true until the next invocation of acquire_mutex() by p_i). Hence, p_i eventually reads j from *AFTER_YOU* and is allowed to enter the critical section.

It follows that a process looses at most one competition with respect to the other process, from which we conclude that the bounded bypass property is satisfied and we have $f(n) = 1$. □

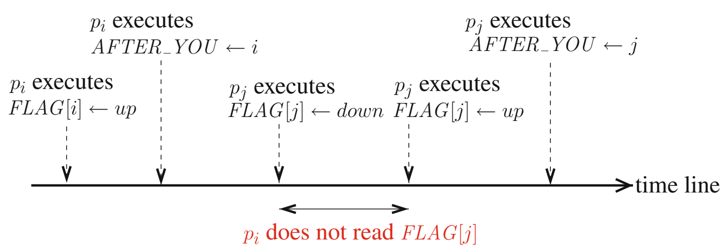

p_i executes $AFTER\_YOU \leftarrow i$
p_j executes $AFTER\_YOU \leftarrow j$
p_i executes $FLAG[i] \leftarrow up$
p_j executes $FLAG[j] \leftarrow down$
p_j executes $FLAG[j] \leftarrow up$
time line
p_i does not read $FLAG[j]$

Fig. 2.7 Bounded bypass property of Peterson's two-process algorithm

Space complexity The space complexity of a mutex algorithm is measured by the number and the size of the atomic registers it uses.

It is easy to see that Peterson's two-process algorithm has a bounded space complexity: there are three atomic registers *FLAG*$[i]$, *FLAG*$[j]$, and *AFTER_YOU*, and the domain of each of them has two values. Hence three atomic bits are sufficient.

2.1.4 Mutex for *n* Processes: Generalizing the Previous Two-Process Algorithm

Description Peterson's mutex algorithm for n processes is described in Fig. 2.8. This algorithm is a simple generalization of the two-process algorithm described in Fig. 2.4. This generalization, which is based on the notion of level, is as follows.

In the two-process algorithm, a process p_i uses a simple SWMR flag *FLAG*$[i]$ whose value is either *down* (to indicate it is not interested in the critical section) or *up* (to indicate it is interested). Instead of this binary flag, a process p_i uses now a multi-valued flag that progresses from a flag level to the next one. This flag, denoted *FLAG_LEVEL*$[i]$, is initialized to 0 (indicating that p_i is not interested in the critical section). It then increases first to level 1, then to level 2, etc., until the level $n-1$,

```
operation acquire_mutex($i$) is
(1) for $\ell$ from 1 to $(n-1)$ do
(2)     $FLAG\_LEVEL[i] \leftarrow \ell$;
(3)     $AFTER\_YOU[\ell] \leftarrow i$;
(4)     wait $(\forall\, k \neq i :\ FLAG\_LEVEL[k] < \ell) \vee (AFTER\_YOU[\ell] \neq i)$
(5) end for;
(6) return()
end operation.

operation release_mutex($i$) is $FLAG\_LEVEL[i] \leftarrow 0$; return() end operation.
```

Fig. 2.8 Peterson's algorithm for n processes (code for p_i)

which allows it to enter the critical section. For $1 \leq x < n-1$, $FLAG\_LEVEL[i] = x$ means that p_i is trying to enter level $x + 1$.

Moreover, to eliminate possible deadlocks at any level ℓ, $0 < \ell < n - 1$ (such as the deadlock that can occur in the algorithm of Fig. 2.3), the processes use a second array of atomic registers $AFTER\_YOU[1..(n-1)]$ such that $AFTER\_YOU[\ell]$ keeps track of the last process that has entered level ℓ.

More precisely, a process p_i executes a **for** loop to progress from one level to the next one, starting from level 1 and finishing at level $n - 1$. At each level the two-process solution is used to block a process (if needed). The predicate that allows a process to progress from level ℓ, $0 < \ell < n - 1$, to level $\ell + 1$ is similar to the one of the two-process algorithm. More precisely, p_i is allowed to progress to level $\ell + 1$ if, from its point of view,

- Either all the other processes are at a lower level (i.e., $\forall\, k \neq i$:$FLAG\_LEVEL[k] < \ell$).
- Or it is not the last one that entered level ℓ (i.e., $AFTER\_YOU[\ell] \neq i$).

Let us notice that the predicate used in the **wait** statement of line 4 involves all but one of the atomic registers $FLAG\_LEVEL[\cdot]$ plus the atomic register $AFTER\_YOU[\ell]$. As these registers cannot be read in a single atomic step, the predicate is repeatedly evaluated asynchronously on each register.

When all processes compete for the critical section, at most $(n - 1)$ processes can concurrently be winners at level 1, $(n - 2)$ processes can concurrently be winners at level 2, and more generally $(n - \ell)$ processes can concurrently be winners at level ℓ. Hence, there is a single winner at level $(n - 1)$.

The code of the operation release_mutex(i) is similar to the one of the two-process algorithm: a process p_i resets $FLAG\_LEVEL[i]$ to its initial value 0 to indicate that it is no longer interested in the critical section.

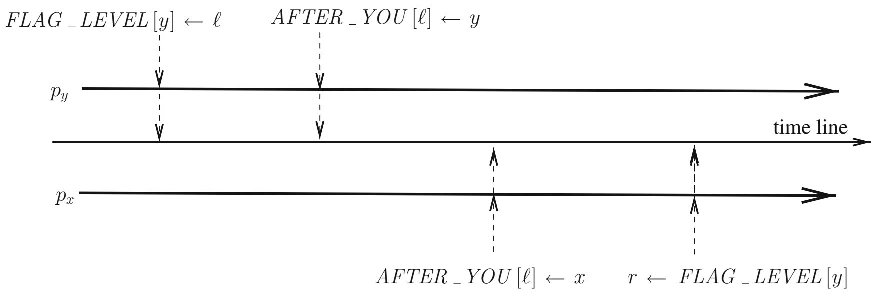
Theorem 2 *The algorithm described in* Fig. 2.8 *satisfies mutual exclusion and starvation-freedom.*

Proof Initially, a process p_i is such that $FLAG\_LEVEL[i] = 0$ and we say that it is at level 0. Let $\ell \in [1..(n - 1)]$. We say that a process p_i has "attained" level ℓ (or, from a global state point of view, "is" at level ℓ) if it has exited the **wait** statement of the ℓth loop iteration. Let us notice that, after it has set its loop index ℓ to $\alpha > 0$ and until it exits the **wait** statement of the corresponding iteration, that process is at level $\alpha - 1$. Moreover, a process that attains level ℓ has also attained the levels ℓ' with $0 \leq \ell' \leq \ell \leq n - 1$ and consequently it is also at these levels ℓ'.

The proof of the mutual exclusion property amounts to showing that at most one process is at level $(n - 1)$. This is a consequence of the following claim when we consider $\ell = n - 1$.

Claim. For ℓ, $0 \leq \ell \leq n - 1$, at most $n - \ell$ processes are at level ℓ.

The proof of this claim is by induction on the level ℓ. The base case $\ell = 0$ is trivial. Assuming that the claim is true up to level $\ell - 1$, i.e., at most $n - (\ell - 1)$



$FLAG\_LEVEL[y] \leftarrow \ell$
$AFTER\_YOU[\ell] \leftarrow y$
p_y
time line
p_x
$AFTER\_YOU[\ell] \leftarrow x$
$r \leftarrow FLAG\_LEVEL[y]$

Fig. 2.9 Total order on read/write operations

processes are simultaneously at level $\ell - 1$, we have to show that at least one process does not progress to level ℓ. The proof is by contradiction: let us assume that $n - \ell + 1$ processes are at level ℓ.

Let p_x be the last process that wrote its identity into $AFTER\_YOU[\ell]$ (hence, $AFTER\_YOU[\ell] = x$). When considering the sequence of read and write operations executed by every process, and the fact that these operations are on atomic registers, this means that, for any of the $n - \ell$ other processes p_y that are at level ℓ, these operations appear as if they have been executed in the following order where the first two operations are issued by p_y while the least two operations are issued by p_x (Fig. 2.9):

1. $FLAG\_LEVEL[y] \leftarrow \ell$ is executed before $AFTER\_YOU[\ell] \leftarrow y$ (sequentiality of p_y)
2. $AFTER\_YOU[\ell] \leftarrow y$ is executed before $AFTER\_YOU[\ell] \leftarrow x$ (assumption: definition of p_x)
3. $AFTER\_YOU[\ell] \leftarrow x$ is executed before $r \leftarrow FLAG\_LEVEL[y]$ (sequentiality of p_x; r is p_x's local variable storing the last value read from $FLAG\_LEVEL[y]$ before p_x exits the **wait** statement at level ℓ).

It follows from this sequence that $r = \ell$. Consequently, as $AFTER\_YOU[\ell] = x$, p_x exited the **wait** statement of the ℓth iteration because $\forall\, k \neq x : FLAG\_LEVEL[k] < \ell$. But this is contradicted by the fact that we had then $FLAG\_LEVEL[y] = \ell$, which concludes the proof of the claim.

The proof of the starvation-freedom property is by induction on the levels starting from level $n - 1$ and proceeding until level 1. The base case $\ell = n - 1$ follows from the previous claim: if there is a process at level $(n - 1)$, it is the only process at that level and it can exit the **for** loop. This process eventually enters the critical section (that, by assumption, it will leave later). The induction assumption is the following: each process that attains a level ℓ' such that $n - 1 \geq \ell' \geq \ell$ eventually enters the critical section.

The rest of the proof is by contradiction. Let us assume that ℓ is such that there is a process (say p_x) that remains blocked forever in the **wait** statement during its ℓth iteration (hence, p_x cannot attain level ℓ). It follows that, each time p_x evaluates the predicate controlling the **wait** statement, we have

$$\big(\exists\, k \neq i : \ FLAG\_LEVEL[k] \geq \ell) \ \wedge \ (AFTER\_YOU[\ell] = x)\big)$$

(let us remember that the atomic registers are read one at a time, asynchronously, and in any order). There are two cases.

- Case 1: There is a process p_y that eventually executes $AFTER\_YOU[\ell] \leftarrow y$.

  As only p_x can execute $AFTER\_YOU[\ell] \leftarrow x$, there is eventually a read of $AFTER\_YOU[\ell]$ that returns a value different from x, and this read allows p_x to progress to level ℓ. This contradicts the assumption that p_x remains blocked forever in the **wait** statement during its ℓth iteration.

- Case 2: No process p_y eventually executes $AFTER\_YOU[\ell] \leftarrow y$.

  The other processes can be partitioned in two sets: the set G that contains the processes at a level greater or equal to ℓ, and the set L that contains the processes at a level smaller than ℓ.

  As the predicate $AFTER\_YOU[\ell] = x$ remains forever true, it follows that no process p_y in L enters the ℓth loop iteration (otherwise p_y would necessarily execute $AFTER\_YOU[\ell] \leftarrow y$, contradicting the case assumption).

  On the other side, due to the induction assumption, all processes in G eventually enter (and later leave) the critical section. When this has occurred, these processes have moved from the set G to the set L and then the predicate $\forall\, k \neq i : \ FLAG\_LEVEL[k] < \ell$ becomes true.

  When this has happened, the values returned by the asynchronous reading of $FLAG\_LEVEL[1..n]$ by p_x allow it to attain level ℓ, which contradicts the assumption that p_x remains blocked forever in the **wait** statement during its ℓth iteration.

In both case the assumption that a process remains blocked forever at level ℓ is contradicted which completes the proof of the induction step and concludes the proof of the starvation-freedom property. $\square$

Starvation-freedom versus bounded bypass The two-process Peterson's algorithm satisfies the bounded bypass liveness property while the n-process algorithm satisfies only starvation-freedom. Actually, starvation-freedom (i.e., finite bypass) is the best liveness property that Peterson's n-process algorithm (Fig. 2.8) guarantees.

This can be shown with a simple example. Let us consider the case $n = 3$. The three processes p_1, p_2, and p_3 invoke simultaneously acquire_mutex(), and the run is such that p_1 wins the competition and enters the critical section. Moreover, let us assume that $AFTER\_YOU[1] = 3$ (i.e., p_3 is the last process that wrote $AFTER\_YOU[1]$) and p_3 blocked at level 1.

Then, after it has invoked release_mutex(), process p_1 invokes acquire_mutex() again and we have consequently $AFTER\_YOU[1] = 1$. But, from that time, p_3 starts an arbitrary long “sleeping” period (this is possible as the processes are asynchronous) and consequently does not read $AFTER\_YOU[1] = 1$ (which would allow it to progress to the second level). Differently, p_2 progresses to the second level and enters the critical section. Later, p_2 first invokes release_mutex() and immediately after invokes acquire_mutex() and updates $AFTER\_YOU[1] = 2$. While p_3 keeps on “sleeping”, p_1 progresses to level 2 and finally enters the critical section. This scenario can be reproduced an arbitrary number of times until p_3 wakes up. When this occurs, p_3 reads from $AFTER\_YOU[1]$ a value different from 3, and consequently progresses to level 2. Hence:

- Due to asynchrony, a “sleeping period” can be arbitrarily long, and a process can consequently lose an arbitrary number of competitions with respect to the other processes,
- But, as a process does not sleep forever, it eventually progresses to the next level.

It is important to notice that, as shown in the proof of the bounded pass property of Theorem 1, this scenario cannot happen when $n = 2$.

Atomic register: size and number It is easy to see that the algorithm uses $2n - 1$ atomic registers. The domain of each of the n registers $FLAG\_LEVEL[i]$ is $[0..(n-1)]$, while the domain of each of the $n-1$ $AFTER\_YOU[\ell]$ registers is $[1..n]$. Hence, in both cases, $\lceil \log_2 n \rceil$ bits are necessary and sufficient for each atomic register.

Number of accesses to atomic registers Let us define the time complexity of a mutex algorithm as the number of accesses to atomic registers for one use of the critical section by a process.

It is easy to see that this cost is finite but not bounded when there is contention (i.e., when several processes simultaneously compete to execute the critical section code).

Differently in a contention-free scenario (i.e., when only one process p_i wants to use the critical section), the number of accesses to atomic registers is $(n - 1)(n + 2)$ in acquire_mutex(i) and one in release_mutex(i).

The case of k-exclusion This is the k-mutual exclusion problem where the critical section code can be concurrently accessed by up to k processes (mutual exclusion corresponds to the case where $k = 1$).

Peterson’s n-process algorithm can easily be modified to solve k-mutual exclusion. The upper bound of the **for** loop (namely $(n-1)$) has simply to be replaced by $(n-k)$. No other statement modification is required. Moreover, let us observe that the size of the array $AFTER\_YOU$ can then be reduced to $[1..(n - k)]$.

2.1.5 Mutex for n Processes: A Tournament-Based Algorithm

Reducing the number of shared memory accesses In the previous n-process mutex algorithm, a process has to compete with the $(n - 1)$ other processes before

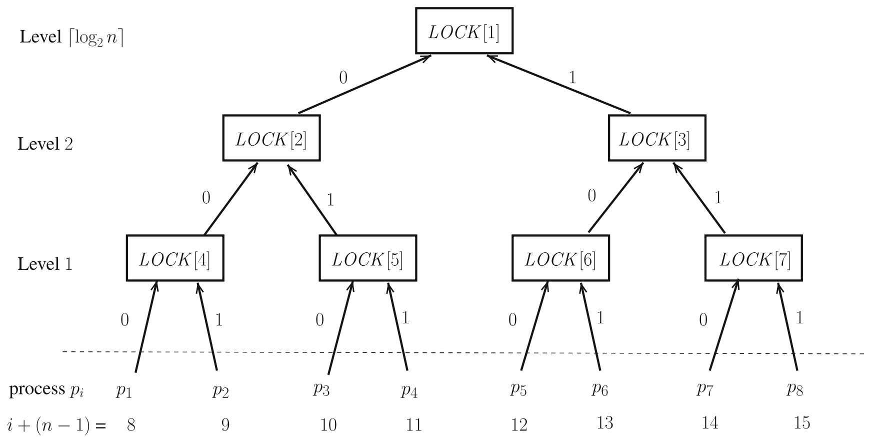

Level $\lceil \log_2 n \rceil$
Level 2
Level 1
$LOCK[1]$
$LOCK[2]$
$LOCK[3]$
$LOCK[4]$
$LOCK[5]$
$LOCK[6]$
$LOCK[7]$
0
1
process p_i
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8
$i + (n-1) =$ 8 9 10 11 12 13 14 15

Fig. 2.10 A tournament tree for n processes

being able to access the critical section. Said differently, it has to execute $n - 1$ loop iterations (eliminating another process at each iteration), and consequently, the cost (measured in number of accesses to atomic registers) in a contention-free scenario is $O(n) \times$ the cost of one loop iteration, i.e., $O(n^2)$. Hence a natural question is the following: Is it possible to reduce this cost and (if so) how?

Tournament tree A simple principle to reduce the number of shared memory accesses is to use a tournament tree. Such a tree is a complete binary tree. To simplify the presentation, we consider that the number of processes is a power of 2, i.e., $n = 2^k$ (hence $k = \log_2 n$). If n is not a power of two, it has to be replaced by $n' = 2^k$ where $k = \lceil \log_2 n \rceil$ (i.e., n' is the smallest power of 2 such that $n' > n$).

Such a tree for $n = 2^3$ processes $p_1, \ldots, p_8$, is represented in Fig. 2.10. Each node of the tree is any two-process starvation-free mutex algorithm, e.g., Peterson's two-process algorithm. It is even possible to associate different two-process mutex algorithms with different nodes. The important common feature of these algorithms is that any of them assumes that it is used by two processes whose identities are 0 and 1.

As we have seen previously, any two-process mutex algorithm implements a lock object. Hence, we consider in the following that the tournament tree is a tree of $(n-1)$ locks and we accordingly adopt the lock terminology. The locks are kept in an array denoted $LOCK[1..(n-1)]$, and for $x \neq y$, $LOCK[x]$ and $LOCK[y]$ are independent objects (the atomic registers used to implement $LOCK[x]$ and the atomic registers used to implement $LOCK[y]$ are different).

The lock $LOCK[1]$ is associated withe root of the tree, and if it is not a leaf, the node associated with the lock $LOCK[x]$ has two children associated with the locks $LOCK[2x]$ and $LOCK[2x+1]$.

According to its identity i, each process p_i starts competing with a single other process p_j to obtain a lock that is a leaf of the tree. Then, when it wins, the process

```
operation acquire_mutex($i$) is
(1)  $node\_id \leftarrow i + (n-1)$;
(2)  for $level$ from $1$ to $k$ do   % $k = \lceil \log_2 n \rceil$ %
(3)     $p\_id[level] \leftarrow node\_id \mod 2$;
(4)     $node\_id \leftarrow \lfloor node\_id/2 \rfloor$;
(5)     $LOCK[node\_id]$.acquire_lock($p\_id[level]$)
(6)  end for;
(7)  return()
end operation.

operation release_mutex($i$) is
(8)  $node\_id \leftarrow 1$;
(9)  for $level$ from $k$ to $1$ do
(10)    $LOCK[node\_id]$.release_lock($p\_id[level]$);
(11)    $node\_id \leftarrow 2 \times node\_id + p\_id[level]$
(12) end for;
(13) return()
end operation.
```

Fig. 2.11 Tournament-based mutex algorithm (code for p_i)

p_i proceeds to the next level of the tree to acquire the lock associated with the node that is the father of the node currently associated with p_i (initially the leaf node associated with p_i). Hence, a process competes to acquire all the locks on the path from the leaf it is associated with until the root node.

As (a) the length of such a path is $\lceil \log_2 n \rceil$ and (b) the cost to obtain a lock associated with a node is $O(1)$ in contention-free scenarios, it is easy to see that the number of accesses to atomic registers in these scenarios is $O(\log_2 n)$ (it is exactly $4\log_2 n$ when each lock is implemented with Peterson's two-process algorithm).

The tournament-based mutex algorithm This algorithm is described in Fig. 2.11. Each process p_i manages a local variable $node\_id$ such that $LOCK[node\_id]$ is the lock currently addressed by p_i and a local array $p\_id[1..k]$ such that $p\_id[\ell]$ is the identity (0 or 1) used by p_i to access $LOCK[node\_id]$ as indicated by the labels on the arrows in Fig. 2.10. (For a process p_i, $p\_id[\ell]$ could be directly computed from the values i and ℓ; a local array is used to simplify the presentation.)

When a process p_i invokes acquire_mutex(i) it first considers that it has successfully locked a fictitious lock object $LOCK[i + (n-1)]$ that can be accessed only by this process (line 1). Process p_i then enters a loop to traverse the tree, level by level, from its starting leaf until the root (lines 2–6). The starting leaf of p_i is associated with the lock $LOCK[\lfloor (i+(n-1))/2 \rfloor]$ (lines 1 and 4). The identity used by p_i to access the lock $LOCK[node\_id]$ (line 5) is computed at line 3 and saved in $p\_id[level]$.

When it invokes release_mutex(i), process p_i releases the k locks it has locked starting from the lock associated with the root ($LOCK[1]$) until the lock associated with its starting leaf $LOCK[\lfloor (i + (n-1))/2 \rfloor]$. When it invokes $LOCK[node\_id]$. release_lock($p\_id[level]$) (line 10), the value of the parameter $p\_id[level]$ is the identity (0 or 1) used by p_i when it locked that object. This identity is also used by p_i to compute the index of the next lock object it has to unlock (line 11).

Theorem 3 *Assuming that each two-process lock object satisfies mutual exclusion and deadlock-freedom (or starvation-freedom), the algorithm described in* Fig. 2.11 *satisfies mutual exclusion and deadlock-freedom (or starvation-freedom).*

Proof The proof of the mutex property is by contradiction. If p_i and p_j $(i \neq j)$ are simultaneously in the critical section, there is a lock object $LOCK[node\_id]$ such that p_i and p_j have invoked acquire_lock() on that object and both have been simultaneously granted the lock. (If there are several such locks, let $LOCK[node\_id]$ be one at the lowest level in the tree.) Due to the specification of the lock object (that grants the lock to a single process identity, namely 0 or 1), it follows that both p_i and p_j have invoked $LOCK[node\_id]$.acquire_lock() with the same identity value (0 or 1) kept in their local variable $p\_id[level]$. But, due to the binary tree structure of the set of lock objects and the way the processes compute $p\_id[level]$, this can only happen if $i = j$ (on the lowest level on which p_i and p_j share a lock), which contradicts our assumption and completes the proof of the mutex property.

The proof of the starvation-freedom (or deadlock-freedom) property follows from the same property of the base lock objects. We consider here only the starvation-freedom property. Let us assume that a process p_i is blocked forever at the object $LOCK[node\_id]$. This means that there is another process p_j that competes infinitely often with p_i for the lock granted by $LOCK[node\_id]$ and wins each time. The proof follows from the fact that, due to the starvation-freedom property of $LOCK[node\_id]$, this cannot happen. □

Remark Let us consider the case where each algorithm implementing an underlying two-process lock object uses a bounded number of bounded atomic registers (which is the case for Peterson's two-process algorithm). In that case, as the tournament-based algorithm uses $(n-1)$ lock objects, it follows that it uses a bounded number of bounded atomic registers.

Let us observe that this tournament-based algorithm has better time complexity than Peterson's n-process algorithm.

2.1.6 A Concurrency-Abortable Algorithm

When looking at the number of accesses to atomic registers issued by acquire_mutex() and release_mutex() for a single use of the critical section in a contention-free scenario, the cost of Peterson's n-process mutual exclusion algorithm is $O(n^2)$ while the cost of the tournament tree-based algorithm is $O(\log_2 n)$. Hence, a natural question is the following: Is it possible to design a *fast* n-process mutex algorithm, where *fast* means that the cost of the algorithm is constant in a contention-free scenario?

The next section of this chapter answers this question positively. To that end, an incremental presentation is adopted. A simple one-shot operation is first presented. Each of its invocations returns a value r to the invoking process, where r is the value $abort$ or the value $commit$. Then, the next section enriches the algorithm implementing this operation to obtain a deadlock-free fast mutual exclusion algorithm due to L. Lamport (1987).

Concurrency-abortable operation A *concurrency-abortable* (also named *contention-abortable* and usually abbreviated *abortable*) operation is an operation that is allowed to return the value $abort$ in the presence of concurrency. Otherwise, it has to return the value $commit$. More precisely, let conc_abort_op() be such an operation. Assuming that each process invokes it at most once (one-shot operation), the set of invocations satisfies the following properties:

- Obligation. If the first process which invokes conc_abort_op() is such that its invocation occurs in a concurrency-free pattern (i.e., no other process invokes conc_abort_op() during its invocation), this process obtains the value $commit$.
- At most one. At most one process obtains the value $commit$.

An n-process concurrency-abortable algorithm Such an algorithm is described in Fig. 2.12. As in the previous algorithms, it assumes that all the processes have distinct identities, but differently from them, the number n of processes can be arbitrary and remains unknown to the processes.

This algorithm uses two MWMR atomic registers denoted X and Y. The register X contains a process identity (its initial value being arbitrary). The register Y contains a process identity or the default value $\perp$ (which is its initial value). It is consequently assumed that these atomic registers are made up of $\lceil \log_2(n+1) \rceil$ bits.

```
operation conc_abort_op($i$) is
(1)  $X \leftarrow i$;
(2)  if ($Y \neq \perp$)
(3)     then return($abort_1$)
(4)     else $Y \leftarrow i$;
(5)          if ($X = i$)
(6)             then return($commit$)
(7)             else return($abort_2$)
(8)          end if
(9)  end if
end operation.
```

Fig. 2.12 An n-process concurrency-abortable operation (code for p_i)

When it invokes conc_abort_op(), a process p_i first deposits its identity in X (line 1) and then checks if the current value of Y is its initial value $\perp$ (line 2). If $Y \neq \perp$, there is (at least) one process p_j that has written into Y. In that case, p_i returns $abort_1$ (both $abort_1$ and $abort_2$ are synonyms of $abort$; they are used only to distinguish the place where the invocation of conc_abort_op() is "aborted"). Returning $abort_1$ means that (from a concurrency point of view) p_i was late: there is another process that wrote into Y before p_i reads it.

If $Y = \perp$, process p_i writes its identity into Y (line 4) and then checks if X is still equal to its identity i (line 5). If this is the case, p_i returns the value $commit$ at line 6 (its invocation of conc_abort_op(i) is then successful). If $X \neq i$, another process p_j has written its identity j into X, overwriting the identity i before p_i reads X at line 5. Hence, there is contention and the value $abort_2$ is returned to p_i (line 7). Returning $abort_2$ means that, among the competing processes that found $y = \perp$, p_i was not the last to have written its name into X.

Remark Let us observe that the only test on Y is $Y \neq \perp$ (line 2). It follows that Y could be replaced by a flag with the associated domain $\{\perp, \top\}$. Line 4 should then be replaced by $Y \leftarrow \top$.

Using such a flag is not considered here because we want to keep the notation consistent with that of the fast mutex algorithm presented below. In the fast mutex algorithm, the value of Y can be either $\perp$ or any process identifier.

Theorem 4 *The algorithm described in Fig. 2.12 guarantees that (a) at most one process obtains the value $commit$ and (b) if the first process that invokes* conc_abort_op() *executes it in a concurrency-free pattern, it obtains the value $commit$.*

Proof The proof of property (b) stated in the theorem is trivial. If the first process (say p_i) that invokes conc_abort_op() executes this operation in a concurrency-free context, we have $Y = \perp$ when it reads Y at line 2 and $X = i$ when it reads X at line 5. It follows that it returns $commit$ at line 6.

Let us now prove property (a), i.e., that no two processes can obtain the value $commit$. Let us assume for the sake of contradiction that a process p_i has invoked conc_abort_op(i) and obtained the value $commit$. It follows from the text of the algorithm that the pattern of accesses to the atomic registers X and Y issued by p_i is the one described in Fig. 2.13 (when not considering the accesses by p_j in that figure). There are two cases.

- Let us first consider the (possibly empty) set Q of processes p_j that read Y at line 2 after this register was written by p_i or another process (let us notice that, due to the atomicity of the registers X and Y, the notion of after/before is well defined). As Y is never reset to $\perp$, it follows that each process $p_j \in Q$ obtains a non-$\perp$ value from Y and consequently executes return($abort_1$) at line 3.

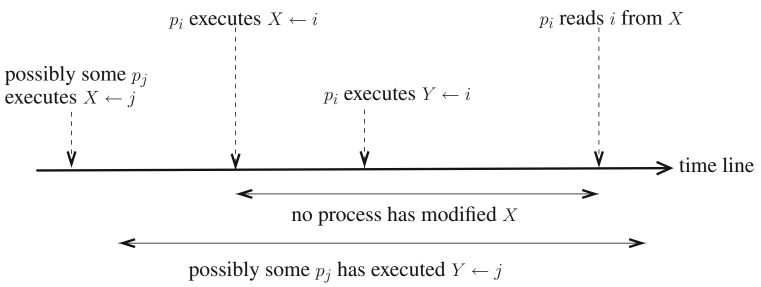

p_i executes $X \leftarrow i$
p_i reads i from X
possibly some p_j executes $X \leftarrow j$
p_i executes $Y \leftarrow i$
time line
no process has modified X
possibly some p_j has executed $Y \leftarrow j$

Fig. 2.13 Access pattern to X and Y for a successful conc_abort_op() invocation by process p_i

- Let us now consider the (possibly empty) set Q' of processes p_j distinct from p_i that read $\perp$ from Y at line 2 concurrently with p_i. Each $p_j \in Q'$ writes consequently its identity j into Y at line 4.

  As p_i has read i from X (line 5), it follows that no process $p_j \in Q'$ has modified X between the execution of line 1 and line 5 by p_i (otherwise p_i would not have read i from X at line 5, see Fig. 2.13). Hence any process $p_j \in Q'$ has written X (a) either before p_i writes i into X or (b) after p_i has read i from X. But, observe that case (b) cannot happen. This is due to the following observation. A process p_k that writes X (at line 1) after p_i has read i from this register (at line 5) necessarily finds $Y \neq \perp$ at line 4 (this is because p_i has previously written i into Y at line 4 before reading i from X at line 5). Consequently, such a process p_k belongs to the set Q and not to the set Q'. Hence, the only possible case is that each $p_j \in Q'$ has written j into X before p_i writes i into X. It follows that p_i is the last process of $Q' \cup \{p_i\}$ which has written its identity into X.

  We conclude from the previous observation that, when a process $p_j \in Q'$ reads X at line 5, it obtains from this register a value different from j and, consequently, its invocation conc_abort_op(j) returns the value $abort_2$, which concludes the proof of the theorem. □

The next corollary follows from the proof of the previous theorem.

Corollary 1 *($Y \neq \perp$) $\Rightarrow$ a process has obtained the value commit or several processes have invoked* conc_abort_op().

Theorem 5 *Whatever the number of processes that invoke* conc_abort_op()*, any of these invocations costs at most four accesses to atomic registers.*

Proof The proof follows from a simple examination of the algorithm. □

Remark: splitter object When we (a) replace the value *commit*, *abort*$_1$, and *abort*$_2$ by *stop*, *right*, and *left*, respectively, and (b) rename the operation conc_abort_op(i) as direction(i), we obtain a one-shot object called a *splitter*. A one-shot object is an object that provides processes with a single operation and each process invokes that operation at most once.

In a run in which a single process invokes direction(), it obtains the value *stop*. In any run, if $m > 1$ processes invoke direction(), at most one process obtains the value *stop*, at most $(m-1)$ processes obtain *right*, and at most $(m-1)$ processes obtain *left*. Such an object is presented in detail in Sect. 5.2.1.

2.1.7 A Fast Mutex Algorithm

Principle and description This section presents L. Lamport's fast mutex algorithm, which is built from the previous one-shot concurrency-abortable operation. More specifically, this algorithm behaves similarly to the algorithm of Fig. 2.12 in contention-free scenarios and (instead of returning *abort*) guarantees the deadlock-freedom liveness property when there is contention.

The algorithm is described in Fig. 2.14. The line numbering is the same as in Fig. 2.12: the lines with the same number are the same in both algorithms, line N0 is new, line N3 replaces line 3, lines N7.1–N7.5 replace line 7, and line N10 is new.

To attain its goal (both fast mutex and deadlock-freedom) the algorithm works as follows. First, each process p_i manages a SWMR flag *FLAG*[i] (initialized to *down*)

```
operation acquire_mutex(i) is
(N0)    FLAG[i] ← up;
(1)     X ← i;
(2)     if (Y ≠ ⊥)
(N3)       then FLAG[i] ← down; wait (Y = ⊥); restart at line N0
(4)        else  Y ← i;
(5)              if (X = i)
(6)                 then return()
(N7.1)              else  FLAG[i] ← down;
(N7.2)                    for each j do wait (FLAG[j] = down) end for;
(N7.3)                    if (Y = i) then return()
(N7.4)                               else  wait (Y = ⊥); restart at line N0
(N7.5)                    end if
(8)              end if
(9)     end if
end operation.

operation release_mutex(i) is
(N10)  Y ← ⊥; FLAG[i] ← down; return()
end operation.
```

Fig. 2.14 Lamport's fast mutex algorithm (code for p_i)

that p_i sets to up to indicate that it is interested in the critical section (line N0). This flag is reset to $down$ when p_i exits the critical section (line N10). As we are about to see, it can be reset to $down$ also in other parts of the algorithm.

According to the contention scenario in which a process p_i returns *abort* in the algorithm of Fig. 2.12, there are two cases to consider, which have been differentiated by the values $abort_1$ and $abort_2$.

- Eliminating $abort_1$ (line N3).

  In this case, as we have seen in Fig. 2.12, process p_i is "late". As captured by Corollary 1, this is because there are other processes that currently compete for the critical section or there is a process inside the critical section. Line 3 of Fig. 2.12 is consequently replaced by the following statements (new line N3):

  – Process p_i first resets its flag to $down$ in order not to prevent other processes from entering the critical section (if no other process is currently inside it).

  – According to Corollary 1, it is useless for p_i to retry entering the critical section while $Y \neq \bot$. Hence, process p_i delays its request for the critical section until $Y = \bot$.

- Eliminating $abort_2$ (lines N7.1–N7.5).

  In this case, as we have seen in the base contention-abortable algorithm (Fig. 2.12), several processes are competing for the critical section (or a process is already inside the critical section). Differently from the base algorithm, one of the competing processes has now to be granted the critical section (if no other process is inside it). To that end, in order not to prevent another process from entering the critical section, process p_i first resets its flag to $down$ (line N7.1). Then, p_i tries to enter the critical section. To that end, it first waits until all flags are down (line N7.2). Then, p_i checks the value of Y (line N7.3). There are two cases:

  – If $Y = i$, process p_i enters the critical section. This is due to the following reason.

    Let us observe that, if $Y = i$ when p_i reads it at line N7.3, then no process has modified Y since p_i set it to the value i at line 4 (the write of Y at line 4 and its reading at line N7.3 follow the same access pattern as the write of X at line 1 and its reading at line 5). Hence, process p_i is the last process to have executed line 4. It then follows that, as it has (asynchronously) seen each flag equal to $down$ (line 7.2), process p_i is allowed to enter the critical section (`return()` statement at line N7.3).

  – If $Y \neq i$, process p_i does the same as what is done at line N3. As it has already set its flag to $down$, it has only to wait until the critical section is released before retrying to enter it (line N7.4). (Let us remember that the only place where Y is reset to $\bot$ is when a process releases the critical section.)

Fast path and slow path The *fast* path to enter the critical section is when p_i executes only the lines N0, 1, 2, 4, 5, and 6. The fast path is open for a process p_i

if it reads i from X at line 5. This is the path that is always taken by a process in contention-free scenarios.

The cost of the fast path is five accesses to atomic registers. As release_mutex() requires two accesses to atomic registers, it follows that the cost of a single use of the critical section in a contention-free scenario is seven accesses to atomic registers.

The *slow* path is the path taken by a process which does not take the fast path. Its cost in terms of accesses to atomic registers depends on the current concurrency pattern.

A few remarks A register $FLAG[i]$ is set to $down$ when p_i exits the critical section (line N10) but also at line N3 or N7.1. It is consequently possible for a process p_k to be inside the critical section while all flags are down. But let us notice that, when this occurs, the value of Y is different from $\perp$, and as already indicated, the only place where Y is reset to $\perp$ is when a process releases the critical section.

When executed by a process p_i, the aim of the **wait** statement at line N3 is to allow any other process p_j to see that p_i has set its flag to $down$. Without such a **wait** statement, a process p_i could loop forever executing the lines N0, 1, 2 and N3 and could thereby favor a livelock by preventing the other processes from seeing $FLAG[i] = down$.

Theorem 6 *Lamport's fast mutex algorithm satisfies mutual exclusion and deadlock-freedom.*

Proof Let us first consider the mutual exclusion property. Let p_i be a process that is inside the critical section. Trivially, we have then $Y \neq \perp$ and p_i returned from acquire_mutex() at line 6 or at line N7.3. Hence, there are two cases. Before considering these two cases, let us first observe that each process (if any) that reads Y after it was written by p_i (or another process) executes line N3: it resets its flag to $down$ and waits until $Y = \perp$ (i.e., at least until p_i exits the critical section, line N10). As the processes that have read a non-$\perp$ value from Y at line 2 cannot enter the critical section, it follows that we have to consider only the processes p_j that have read $\perp$ from Y at line 2.

- Process p_i has executed return() at line 6.

  In this case, it follows from a simple examination of the text of the algorithm that $FLAG[i]$ remains equal to up until p_i exits the critical section and executes line N10.

  Let us consider a process p_j that has read $\perp$ from Y at line 2. As process p_i has executed line 6, it was the last process (among the competing processes which read $\perp$ from Y) to have written its identity into X (see Fig. 2.13) and consequently p_j cannot read j from X. As $X \neq j$ when p_j reads X at line 5, it follows that process p_j executes the lines N7.1–N7.5. When it executes line N7.2, p_j remains blocked until p_i resets its flag to $down$, but as we have seen, p_i does so only when it exits the critical section. Hence, p_j cannot be inside the critical section simultaneously with p_i. This concludes the proof of the first case.

- Process p_i has executed return() at line N7.3.

  In this case, the predicate $Y = i$ allowed p_i to enter the critical section. Moreover, the atomic register Y has not been modified during the period starting when it was assigned the identity i at line 4 by p_i and ending at the time at which p_i read it at line N7.3. It follows that, among the processes that read $\perp$ from Y (at line 2), p_i is the last one to have updated Y.

  Let us observe that $X \neq j$, otherwise p_j would have entered the critical section at line 6, and in that case (as shown in the previous item) p_i could not have entered the critical section.

  As $Y = i$, it follows from the test of line N7.3 that p_j executes line N7.4 and consequently waits until $Y = \perp$. As Y is set to $\perp$ only when a process exits the critical section (line N10), it follows that p_j cannot be inside the critical section simultaneously with p_i, which concludes the proof of the second case.

To prove the deadlock-freedom property, let us assume that there is a non-empty set of processes that compete to enter the critical section and, from then on, no process ever executes return() at line 6 or line N 7.3. We show that this is impossible.

As processes have invoked acquire_mutex() and none of them executes line 6, it follows that there is among them at least one process p_x that has executed first line N0 and line 1 (where it assigned its identity x to X) and then line N3. This assignment of x to X makes the predicate of line 5 false for the processes that have obtained $\perp$ from Y. It follows that the flag of these processes p_x are eventually reset to *down* and, consequently, these processes cannot entail a permanent blocking of any other process p_i which executes line N7.2.

When the last process that used the critical section released it, it reset Y to $\perp$ (if there is no such process, we initially have $Y = \perp$). Hence, among the processes that have invoked acquire_mutex(), at least one of them has read $\perp$ from Y. Let Q be this (non-empty) set of processes. Each process of Q executes lines N7.1–N7.5 and, consequently, eventually resets its flag to *down* (line N7.1). Hence, the predicate evaluated in the **wait** statement at line N7.2 eventually becomes satisfied and the processes of Q which execute the lines N7.1–N7.5 eventually check at line N7.3 if the predicate $Y = i$ is satisfied. (Due to asynchrony, it is possible that the predicate used at N7.2 is never true when evaluated by some processes. This occurs for the processes of Q which are slow while another process of Q has entered the critical section and invoked acquire_mutex() again, thereby resetting its flag to *up*. The important point is that this can occur only if some process entered the critical section, hence when there is no deadlock.)

As no process is inside the critical section and the number of processes is finite, there is a process p_j that was the last process to have modified Y at line 4. As (by assumption) p_j has not executed return() at line 6, it follows that it executes line N7.3 and, finding $Y = j$, it executes return(), which contradicts our assumption and consequently proves the deadlock-freedom property. □

2.1.8 Mutual Exclusion in a Synchronous System

Synchronous system Differently from an asynchronous system (in which there is no time bound), a synchronous system is characterized by assumptions on the speed of processes. More specifically, there is a bound Δ on the speed of processes and this bound is known to them (meaning that Δ can be used in the code of the algorithms). The meaning of Δ is the following: two consecutive accesses to atomic registers by a process are separated by at most Δ time units.

Moreover, the system provides the processes with a primitive delay(d), where d is a positive duration, which stops the invoking process for a finite duration greater than d. The synchrony assumption applies only to consecutive accesses to atomic registers that are not separated by a delay() statement.

Fischer's algorithm A very simple mutual exclusion algorithm (due to M. Fischer) is described in Fig. 2.15. This algorithm uses a single atomic register X (initialized to $\perp$) that, in addition to $\perp$, can contain any process identity.

When a process p_i invokes acquire_mutex(i), it waits until $X = \perp$. Then it writes its identity into X (as before, it is assumed that no two processes have the same identity) and invokes delay(Δ). When it resumes its execution, it checks if X contains its identity. If this is the case, its invocation acquire_mutex(i) terminates and p_i enters the critical section. If $X \neq i$, it re-executes the loop body.

Theorem 7 *Let us assume that the number of processes is finite and all have distinct identities. Fischer's mutex algorithm satisfies mutual exclusion and deadlock-freedom.*

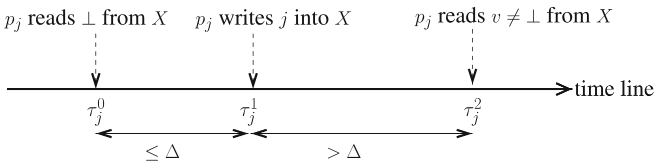
Proof To simplify the statement of the proof we consider that each access to an atomic register is instantaneous. (Considering that such accesses take bounded duration is straightforward.)

Proof of the mutual exclusion property. Assuming that, at some time, processes invoke acquire_mutex(), let C be the subset of them whose last read of X returned $\perp$. Let us observe that the ones that read a non-$\perp$ value from X remain looping in the

```
operation acquire_mutex(i) is
 (1)  repeat wait ($X = \perp$);
 (2)          $X \leftarrow i$;
 (3)          delay($\Delta$)
 (4)  until ($X = i$) end repeat;
 (5)  return()
end operation.

operation release_mutex(i) is
 (6)  $X \leftarrow \perp$; return()
end operation.
```

Fig. 2.15 Fischer's synchronous mutex algorithm (code for p_i)



p_j reads $\perp$ from X
p_j writes j into X
p_j reads $v \neq \perp$ from X
time line
τ_j^0
τ_j^1
τ_j^2
$\leq \Delta$
$> \Delta$

Fig. 2.16 Accesses to X by a process p_j

wait statement at line 1. By assumption, C is finite. Due to the atomicity of the register X and the fact that all processes in C write into X, there is a last process (say p_i) that writes its identity into X.

Given any process p_j of C let us define the following time instants (Fig. 2.16):

- τ_j^0 = time at which p_j reads the value $\perp$ from X (line 1),
- τ_j^1 = time at which p_j writes its identity j into X (line 2), and
- τ_j^2 = time at which p_j reads X (line 4) after having executed the delay(Δ) statement (line 3).

Due to the synchrony assumption and the delay() statement we have $\tau_j^1 \leq \tau_j^0 + \Delta$ (P1) and $\tau_j^2 > \tau_j^1 + \Delta$ (P2). We show that, after p_i has written i into X, this register remains equal to i until p_i resets it to $\perp$ (line 6) and any process p_j of C reads i from X at line 4 from which follows the mutual exclusion property. This is the consequence of the following observations:

1. $\tau_j^1 + \Delta < \tau_j^2$ (property P2),
2. $\tau_i^0 < \tau_j^1$ (otherwise p_i would not have read $\perp$ from X at line 1),
3. $\tau_i^0 + \Delta < \tau_j^1 + \Delta$ (adding Δ to both sides of the previous line),
4. $\tau_i^1 \leq \tau_i^0 + \Delta < \tau_j^1 + \Delta < \tau_j^2$ (from P1 and the previous items 1 and 3).

It then follows from the fact that p_i is the last process which wrote into X and $\tau_j^2 > \tau_i^1$ that p_j reads i from X at line 4 and consequently does enter the **repeat** loop again and waits until $X = \perp$. The mutual exclusion property follows.

Proof of the deadlock-freedom property. This is an immediate consequence of the fact that, among the processes that have concurrently invoked the operation acquire_mutex(), the last process that writes X (p_i in the previous reasoning) reads its own identity from X at line 4. $\square$

Short discussion The main property of this algorithm is its simplicity. Moreover, its code is independent of the number of processes.

2.2 Mutex Based on Specialized Hardware Primitives

The previous section presented mutual exclusion algorithms based on atomic read/write registers. These algorithms are important because understanding their design and their properties provides us with precise knowledge of the difficulty and subtleties that have to be addressed when one has to solve synchronization problems. These algorithms capture the essence of synchronization in a read/write shared memory model.

Nearly all shared memory multiprocessors propose built-in primitives (i.e., atomic operations implemented in hardware) specially designed to address synchronization issues. This section presents a few of them (the ones that are the most popular).

2.2.1 Test&Set, Swap, and Compare&Swap

The test&set()/reset() primitives This pair of primitives, denoted test&set() and reset(), is defined as follows. Let X be a shared register initialized to 1.

- X.test&set() sets X to 0 and returns its previous value.
- X.reset() writes 1 into X (i.e., resets X to its initial value).

Given a register X, the operations X.test&set() and X.reset() are atomic. As we have seen, this means that they appear as if they have been executed sequentially, each one being associated with a point of the time line (that lies between its beginning and its end).

As shown in Fig. 2.17 (where r is local variable of the invoking process), solving the mutual exclusion problem (or equivalently implementing a lock object), can be easily done with a test&set register. If several processes invoke simultaneously X.test&set(), the atomicity property ensures that one and only of them wins (i.e., obtains the value 1 which is required to enter the critical section). Releasing the critical section is done by resetting X to 1 (its initial value). It is easy to see that this implementation satisfies mutual exclusion and deadlock-freedom.

The swap() primitive Let X be a shared register. The primitive denoted X.swap(v) atomically assigns v to X and returns the previous value of X.

Mutual exclusion can be easily solved with a swap register X. Such an algorithm is depicted in Fig. 2.18 where X is initialized to 1. It is assumed that the invoking process

```
operation acquire_mutex() is
    repeat $r \leftarrow X$.test&set() until $(r = 1)$ end repeat;
    return()
end operation.

operation release_mutex() is
    $X$.reset(); return()
end operation.
```

Fig. 2.17 Test&set-based mutual exclusion

```
operation acquire_mutex() is
        $r \leftarrow 0$;
        repeat $r \leftarrow X$.swap($r$) until $(r = 1)$ end repeat;
        return()
end operation.

operation release_mutex() is
        $X$.swap($r$); return()
end operation.
```

Fig. 2.18 Swap-based mutual exclusion

does not modify its local variable r between acquire_mutex() and release_mutex() (or, equivalently, that it sets r to 1 before invoking release_mutex()). The test&set-based algorithm and the swap-based algorithm are actually the very same algorithm.

Let r_i be the local variable used by each process p_i. Due to the atomicity property and the "exchange of values" semantics of the swap() primitive, it is easy to see the swap-based algorithm is characterized by the invariant $X + \Sigma_{1 \le i \le n} r_i = 1$.

The compare&swap() primitive Let X be a shared register and old and new be two values. The semantics of the primitive X.compare&swap(old, new), which returns a Boolean value, is defined by the following code that is assumed to be executed atomically.

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

The primitive compare&swap() is an atomic conditional write; namely, the write of new into X is executed if and only if $X = old$. Moreover, a Boolean value is returned that indicates if the write was successful. This primitive (or variants of it) appears in Motorola 680x0, IBM 370, and SPARC architectures. In some variants, the primitive returns the previous value of X instead of a Boolean.

A compare&swap-based mutual exclusion algorithm is described in Fig. 2.19 in which X is an atomic compare&swap register initialized to 1. (no-op means "no operation".) The **repeat** statement is equivalent to **wait** (X.compare&swap $(1, 0)$); it is used to stress the fact that it is an active waiting. This algorithm is nearly the same as the two previous ones.

2.2.2 From Deadlock-Freedom to Starvation-Freedom

A problem due to asynchrony The previous primitives allow for the (simple) design of algorithms that ensure mutual exclusion and deadlock-freedom. Said differently, these algorithms do not ensure starvation-freedom.

```
operation acquire_mutex() is
    repeat $r \leftarrow X$.compare&swap(1, 0) until ($r$) end repeat;
    return()
end operation.

operation release_mutex() is
    $X \leftarrow 1$; return()
end operation.
```

Fig. 2.19 Compare&swap-based mutual exclusion

As an example, let us consider the test&set-based algorithm (Fig. 2.17). It is possible that a process p_i executes X.test&set() infinitely often and never obtains the winning value 1. This is a simple consequence of asynchrony: if, infinitely often, other processes invoke X.test&set() concurrently with p_i (some of these processes enter the critical section, release it, and re-enter it, etc.), it is easy to construct a scenario in which the winning value is always obtained by only a subset of processes not containing p_i. If X infinitely often switches between 1 to 0, an infinite number of accesses to X does not ensure that one of these accesses obtains the value 1.

From deadlock-freedom to starvation-freedom Considering that we have an underlying lock object that satisfies mutual exclusion and deadlock-freedom, this section presents an algorithm that builds on top of it a lock object that satisfies the starvation-freedom property. Its principle is simple: it consists in implementing a round-robin mechanism that guarantees that no request for the critical section is delayed forever. To that end, the following underlying objects are used:

- The underlying deadlock-free lock is denoted $LOCK$. Its two operations are $LOCK$.acquire_lock(i) and $LOCK$.release_lock(i), where i is the identity of the invoking process.
- An array of SWMR atomic registers denoted $FLAG[1..n]$ (n is the number of processes, hence this number has to be known). For each i, $FLAG[i]$ is initialized to $down$ and can be written only by p_i. In a very natural way, process p_i sets $FLAG[i]$ to up when it wants to enter the critical section and resets it to $down$ when it releases it.
- $TURN$ is an MWMR atomic register that contains the process which is given priority to enter the critical section. Its initial value is any process identity.

  Let us notice that accessing $FLAG[TURN]$ is not an atomic operation. A process p_i has first to obtain the value v of $TURN$ and then address $FLAG[v]$. Moreover, due to asynchrony, between the read by p_i first of $TURN$ and then of $FLAG[v]$, the value of $TURN$ has possibly been changed by another process p_j.

The behavior of a process p_i is described in Fig. 2.20. It is as follows. The processes are considered as defining a logical ring $p_i, p_{i+1}, \ldots, p_n, p_1, \ldots, p_i$. At any time,

```
operation acquire_mutex($i$) is
(1)   $FLAG[i] \leftarrow up$;
(2)   wait $(TURN = i) \vee (FLAG[TURN] = down)$ ;
(3)   $LOCK$.acquire_lock($i$);
(4)   return()
end operation.

operation release_mutex($i$) is
(5)   $FLAG[i] \leftarrow down$;
(6)   if $(FLAG[TURN] = down)$ then $TURN \leftarrow (TURN \mod n) + 1$ end if;
(7)   $LOCK$.release_lock($i$);
(8)   return()
end operation.
```

Fig. 2.20 From deadlock-freedom to starvation-freedom (code for p_i)

the process p_{TURN} is the process that has priority and $p_{(TURN \mod n)+1}$ is the next process that will have priority.

- When a process p_i invokes acquire_mutex(i) it first raises its flag to inform the other processes that it is interested in the critical section (line 1). Then, it waits (repeated checks at line 2) until it has priority (predicate $TURN = i$) or the process that is currently given the priority is not interested (predicate $FLAG[TURN] = down$). Finally, as soon as it can proceed, it invokes $LOCK$.acquire_lock(i) in order to obtain the underlying lock (line 3). (Let us remember that reading $FLAG[TURN]$ requires two shared memory accesses.)
- When a process p_i invokes release_mutex(i), it first resets its flag to $down$ (line 5). Then, if (from p_i's point view) the process that is currently given priority is not interested in the critical section (i.e., the predicate $FLAG[TURN] = down$ is satisfied), then p_i makes $TURN$ progress to the next process (line 6) on the ring before releasing the underlying lock (line 7).

Remark 1 Let us observe that the modification of $TURN$ by a process p_i is always done in the critical section (line 6). This is due to the fact that p_i modifies $TURN$ after it has acquired the underlying mutex lock and before it has released it.

Remark 2 Let us observe that a process p_i can stop waiting at line 2 because it finds $TURN = i$ while another process p_j increases $TURN$ to $((i + 1) \mod n)$ because it does not see that $FLAG[i]$ has been set to up. This situation is described in Fig. 2.21.

Theorem 8 *Assuming that the underlying mutex lock LOCK is deadlock-free, the algorithm described in* Fig. 2.20 *builds a starvation-free mutex lock.*

Proof We first claim that, if at least one process invokes acquire_mutex(), then at least one process invokes $LOCK$.acquire_lock() (line 3) and enters the critical section.

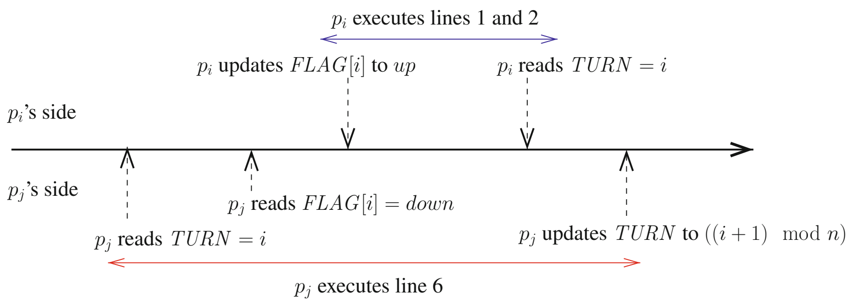

p_i executes lines 1 and 2
p_i updates $FLAG[i]$ to up
p_i reads $TURN = i$
p_i's side
p_j's side
p_j reads $FLAG[i] = down$
p_j reads $TURN = i$
p_j updates $TURN$ to $((i+1) \mod n)$
p_j executes line 6

Fig. 2.21 A possible case when going from deadlock-freedom to starvation-freedom

Proof of the claim. Let us first observe that, if processes invoke $LOCK$.acquire_lock(), one of them enters the critical section (this follows from the fact that the lock is deadlock-free). Hence, X being the non-empty set of processes that invoke acquire_mutex(), let us assume by contradiction that no process of X terminates the **wait** statement at line 2. It follows from the waiting predicate that $TURN \notin X$ and $FLAG[TURN] = up$. But, $FLAG[TURN] = up$ implies $TURN \in X$, which contradicts the previous waiting predicate and concludes the proof of the claim.

Let p_i be a process that has invoked acquire_mutex(). We have to show that it enters the critical section. Due to the claim, there is a process p_k that holds the underlying lock. If p_k is p_i, the theorem follows, hence let $p_k \neq p_i$. When p_k exits the critical section it executes line 6. Let $TURN = j$ when p_k reads it. We consider two cases:

1. $FLAG[j] = up$. Let us observe that p_j is the only process that can write into $FLAG[j]$ and that it will do so at line 5 when it exits the critical section. Moreover, as $TURN = j$, p_j is not blocked at line 2 and consequently invokes $LOCK$.acquire_lock() (line 3).

   We first show that eventually p_j enters the critical section. Let us observe that all the processes which invoke acquire_mutex() after $FLAG[j]$ was set to up and $TURN$ was set to j remain blocked at line 2 (Observation OB). Let Y be the set of processes that compete with p_j for the lock with $y = |Y|$. We have $0 \leq y \leq n-1$. It follows from observation OB and the fact that the lock is deadlock-free that the number of processes that compete with p_j decreases from y to $y-1$, $y-2$, etc., until p_j obtains the lock and executes line 5 (in the worst case, p_j is the last of the y processes to obtain the lock).

   If p_i is p_j or a process that has obtained the lock before p_j, the theorem follows from the previous reasoning. Hence, let us assume that p_i has not obtained the lock. After p_j has obtained the lock, it eventually executes lines 5 and 6. As $TURN = j$ and p_j sets $FLAG[j]$ to $down$, it follows that p_j updates the register $TURN$ to $\ell = (j \mod n)+1$. The previous reasoning, where k and j are replaced by j and ℓ, is then applied again.

2. $\mathit{FLAG}[j] = \mathit{down}$. In this case, p_k updates *TURN* to $\ell = (j \mod n) + 1$. If $\ell = i$, the previous reasoning (where p_j is replaced by p_i) applies and it follows that p_i obtains the lock and enters the critical section.

   If $\ell \neq i$, let $p_{k'}$ be the next process that enters the critical section (due to the claim, such a process does exist). Then, the same reasoning as in case 1 applies, where k is replaced by k'.

As no process is skipped when *TURN* is updated when processes invoke release_mutex(), it follows from the combination of case 1 and case 2 that eventually case 1 where $p_j = p_i$ applies and consequently p_i obtains the deadlock-free lock. $\square$

Fast starvation-free mutual exclusion Let us consider the case where a process p_i wants to enter the critical section, while no other process is interested in entering it. We have the following:

- The invocation of acquire_mutex(i) requires at most three accesses to the shared memory: one to set the register $\mathit{FLAG}[i]$ to up, one to read *TURN* and save it in a local variable turn, and one to read $\mathit{FLAG}[\mathit{turn}]$.
- Similarly, the invocation by p_i of release_mutex(i) requires at most four accesses to the shared memory: one to reset $\mathit{FLAG}[i]$ to down, one to read *TURN* and save it in a local variable turn, one to read $\mathit{FLAG}[\mathit{turn}]$, and a last one to update *TURN*.

It follows from this observation that the stacking of the algorithm of Fig. 2.20 on top of the algorithm described in Fig. 2.14 (Sect. 2.1.7), which implements a deadlock-free fast mutex lock, provides a fast starvation-free mutex algorithm.

2.2.3 Fetch&Add

Let X be a shared register. The primitive X.fetch&add() atomically adds 1 to X and returns the new value. (In some variants the value that is returned is the previous value of X. In other variants, a value c is passed as a parameter and, instead of being increased by 1, X becomes $X + c$.)

Such a primitive allows for the design of a simple starvation-free mutex algorithm. Its principle is to use a fetch&add atomic register to generate tickets with consecutive numbers and to allow a process to enter the critical section when its ticket number is the next one to be served.

An algorithm based on this principle is described in Fig. 2.22. The variable *TICKET* is used to generate consecutive ticket values, and the variable *NEXT* indicates the next winner ticket number. *TICKET* is initialized to 0, while *NEXT* is initialized to 1.

When it invokes acquire_mutex(), a process p_i takes the next ticket, saves it in its local variable $\mathit{my\_turn}$, and waits until its turn occurs, i.e., until ($\mathit{my\_turn} = \mathit{NEXT}$). An invocation of release_mutex() is a simple increase of the atomic register *NEXT*.

```
**operation** acquire_mutex() **is**
    $my\_turn \leftarrow TICKET$.fetch&add();
    **repeat** no-op **until** $(my\_turn = NEXT)$ **end repeat**;
    return()
**end operation**.

**operation** release_mutex() **is**
    $NEXT \leftarrow NEXT + 1$; return()
**end operation**.
```

Fig. 2.22 Fetch&add-based mutual exclusion

Let us observe that, while *NEXT* is an atomic MWMR register, the operation *NEXT* $\leftarrow$ *NEXT* $+ 1$ is not atomic. It is easy to see that no increase of *NEXT* can be missed. This follows from the fact that the increase statement *NEXT* $\leftarrow$ *NEXT* $+ 1$ appears in the operation release_mutex(), which is executed by a single process at a time.

The mutual exclusion property follows from the uniqueness of each ticket number, and the starvation-freedom property follows from the fact that the ticket numbers are defined from a sequence of consecutive known values (here the increasing sequence of positive integers).

2.3 Mutex Without Atomicity

This section presents two mutex algorithms which rely on shared read/write registers weaker than read/write atomic registers. In that sense, they implement atomicity without relying on underlying atomic objects.

2.3.1 Safe, Regular, and Atomic Registers

The algorithms described in this section rely on *safe* registers. As shown here, safe registers are the weakest type of shared registers that we can imagine while being useful, in the presence of concurrency.

As an atomic register, a safe register (or a regular register) *R* provides the processes with a write operation denoted R.write(v) (or $R \leftarrow v$), where v is the value that is written and a read operation R.read() (or $local \leftarrow R$, where $local$ is a local variable of the invoking process). Safe, regular and atomic registers differ in the value returned by a read operation invoked in the presence of concurrent write operations.

Let us remember that the domain of a register is the set of values that it can contain. As an example, the domain of a binary register is the set {0, 1}.

SWMR safe register An SWMR *safe* register is a register whose read operation satisfies the following properties (the notion of an MWMR safe register will be introduced in Sect. 2.3.3):

- A read that is not concurrent with a write operation (i.e., their executions do not overlap) returns the current value of the register.
- A read that is concurrent with one (or several consecutive) write operation(s) (i.e., their executions do overlap) returns *any* value that the register can contain.

It is important to see that, in the presence of concurrent write operations, a read can return a value that has never been written. The returned value has only to belong to the register domain. As an example, let the domain of a safe register R be $\{0, 1, 2, 3\}$. Assuming that $R = 0$, let $R.\mathsf{write}(2)$ be concurrent with a read operation. This read can return 0, 1, 2, or 3. It cannot return 4, as this value is not in the domain of R, but can return the value 3, which has never been written.

A *binary safe* register can be seen as modeling a flickering bit. Whatever its previous value, the value of the register can flicker during a write operation and stabilizes to its final value only when the write finishes. Hence, a read that overlaps with a write can arbitrarily return either 0 or 1.

SWMR regular register An SWMR *regular* register is an SWMR safe register that satisfies the following property. This property addresses read operations in thee presence of concurrency. It replaces the second item of the definition of a safe register.

- A read that is concurrent with one or several write operations returns the value of the register before these writes or the value written by any of them.

An example of a regular register R (whose domain is the set $\{0, 1, 2, 3, 4\}$) written by a process p_1 and read by a process p_2 is described in Fig. 2.23. As there is no concurrent write during the first read by p_2, this read operation returns the current value of the register R, namely 1. The second read operation is concurrent with three write operations. It can consequently return any value in $\{1, 2, 3, 4\}$. If the register was only safe, this second read could return any value in $\{0, 1, 2, 3, 4\}$.

Atomic register The notion of an atomic register was defined in Sect. 2.1.1. Due to the total order on all its operations, an atomic register is more constrained (i.e., stronger) than a regular register.

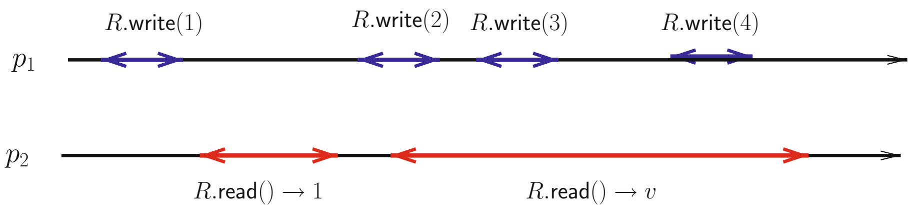


Fig. 2.23 An execution of a regular register

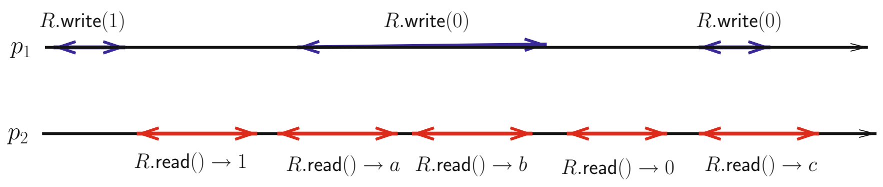


Fig. 2.24 An execution of a register

Table 2.1 Values returned by safe, regular and atomic registers

Value returned	a	b	c	Number of correct executions
Safe	1/0	1/0	1/0	8
Regular	1/0	1/0	0	4
Atomic	1	1/0	0	3
Atomic	0	0	0	

To illustrate the differences between safe, regular, and atomic, Fig. 2.24 presents an execution of a binary register R and Table 2.1 describes the values returned by the read operations when the register is safe, regular, and atomic. The first and third read by p_2 are issued in a concurrency-free context. Hence, whatever the type of the register, the value returned is the current value of the register R.

- If R is safe, as the other read operations are concurrent with a write operation, they can return any value (i.e., 0 or 1 as the register is binary). This is denoted 0/1 in Table 2.1.

  It follows that there are eight possible correct executions when the register R is safe for the concurrency pattern depicted in Fig. 2.24.

- If R is regular, each of the values a and b returned by the read operation which is concurrent with R.write(0) can be 1 (the value of R before the read operation) or 0 (the value of R that is written concurrently with the read operation).

  Differently, the value c returned by the last read operation can only be 0 (because the value that is written concurrently does not change the value of R).

  It follows that there are only four possible correct executions when the register R is regular.

- If R is atomic, there are only three possible executions, each corresponding to a correct sequence of read and write invocations ("correct" means that the sequence respects the real-time order of the invocations and is such that each read invocation returns the value written by the immediately preceding write invocation).

2.3.2 The Bakery Mutex Algorithm

Principle of the algorithm The mutex algorithm presented in this section is due to L. Lamport (1974) who called it the mutex *bakery algorithm*. It was the first algorithm ever designed to solve mutual exclusion on top of non-atomic registers, namely on top of SWMR safe registers. The principle that underlies its design (inspired from bakeries where a customer receives a number upon entering the store, hence the algorithm name) is simple. When a process p_i wants to acquire the critical section, it acquires a number x that defines its priority, and the processes enter the critical section according to their current priorities.

As there are no atomic registers, it is possible that two processes obtain the same number. A simple way to establish an order for requests that have the same number consists in using the identities of the corresponding processes. Hence, let a pair $\langle x, i\rangle$ define the identity of the current request issued by p_i. A total order is defined for the requests competing for the critical section as follows, where $\langle x, i\rangle$ and $\langle y, j\rangle$ are the identities of two competing requests; $\langle x, i\rangle < \langle y, j\rangle$ means that the request identified by $\langle x, i\rangle$ has priority over the request identified by $\langle y, j\rangle$ where "<" is defined as the lexicographical ordering on pairs of integers, namely

$$\langle x, i\rangle < \langle y, j\rangle \equiv (x < y) \vee ((x = y) \wedge (i < j)).$$

Description of the algorithm Two SWMR safe registers, denoted $FLAG[i]$ and $MY\_TURN[i]$, are associated with each process p_i (hence these registers can be read by any process but written only by p_i).

- $MY\_TURN[i]$ (which is initialized to 0 and reset to that value when p_i exits the critical section) is used to contain the priority number of p_i when it wants to use the critical section. The domain of $MY\_TURN[i]$ is the set of non-negative integers.

- $FLAG[i]$ is a binary control variable whose domain is $\{down, up\}$. Initialized to $down$, it is set to up by p_i while it computes the value of its priority number $MY\_TURN[i]$.

  The sequence of values taken by $FLAG[i]$ is consequently the regular expression $down(up, down)^*$. The reader can verify that a binary safe register whose write operations of $down$ and up alternate behaves as a regular register.

The algorithm of a process p_i is described in Fig. 2.25. When it invokes acquire_mutex(), process p_i enters a "doorway" (lines 1–3) in which it computes its turn number $MY\_TURN[i]$ (line 2). To that end it selects a number greater than all $MY\_TURN[j]$, $1 \leq j \leq n$. It is possible that p_i reads some $MY\_TURN[j]$ while it is written by p_j. In that case the value obtained from $MY\_TURN[j]$ can be any value. Moreover, a process informs the other processes that it is computing its turn value by raising its flag before this computation starts (line 1) and resetting it to $down$ when it has finished (line 3). Let us observe that a process is never delayed while in the doorway, which means no process can direct another process to wait in the doorway.

```
operation acquire_mutex($i$) is
(1)  $FLAG[i] \leftarrow up$;
(2)  $MY\_TURN[i] \leftarrow \max(MY\_TURN[1], \ldots, MY\_TURN[n]) + 1$;
(3)  $FLAG[i] \leftarrow down$;
(4)  for each $j \in \{1, \ldots, n\} \setminus \{i\}$ do
(5)      wait ($FLAG[j] = down$);
(6)      wait (($MY\_TURN[j] = 0) \vee \langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$)
(7)  end for;
(8)  return()
end operation.

operation release_mutex($i$) is
(9)  $MY\_TURN[i] \leftarrow 0$; return()
end operation.
```

Fig. 2.25 Lamport's bakery mutual exclusion algorithm

After it has computed its turn value, a process p_i enters a "waiting room" (lines 4–7) which consists of a **for** loop with one loop iteration per process p_j. There are two cases:

- If p_j does not want to enter the critical section, we have $FLAG[j] = down \wedge MY\_TURN[j] = 0$. In this case, p_i proceeds to the next iteration without being delayed by p_j.
- Otherwise, p_i waits until $FLAG[j] = down$ (i.e., until p_j has finished to compute its turn, line 5) and then waits until either p_j has exited the critical section (predicate $MY\_TURN[j] = 0$) or p_i's current request has priority over p_j's one (predicate $\langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$).

  When p_i has priority with respect to each other process (these priorities being checked in an arbitrary order, one after the other) it enters the critical section (line 8).

Finally, when it exits the critical section, the only thing a process p_i has to do is to reset $MY\_TURN[i]$ to 0 (line 9).

Remark: process crashes Let us consider the case where a process may crash (i.e., stop prematurely). It is easy to see that the algorithm works despite this type of failure if, after a process p_i has crashed, its two registers $FLAG[i]$ and $MY\_TURN[i]$ are eventually reset to their initial values. When this occurs, the process p_i is considered as being no longer interested in the critical section.

A first in first out (FIFO) order As already indicated, the priority of a process p_i over a process p_j is defined from the identities of their requests, namely the pairs $\langle MY\_TURN[i], i \rangle$ and $\langle MY\_TURN[j], j \rangle$. Moreover, let us observe that it is not possible to predict the values of these pairs when p_i and p_j compute concurrently the values of $MY\_TURN[i]$ and $MY\_TURN[j]$.

Let us consider two processes p_i and p_j that have invoked acquire_mutex() and where p_i has executed its doorway part (line 2) before p_j has started executing its doorway part. We will see that the algorithm guarantees a FIFO order property defined as follows: p_i terminates its invocation of acquire_mutex() (and consequently enters the critical section) before p_j. This FIFO order property is an instance of the bounded bypass liveness property with $f(n) = n - 1$.

Definitions The following time instant definitions are used in the proof of Theorem 9. Let p_x be a process. Let us remember that, as the read and write operations on the registers are not atomic, they cannot be abstracted as having been executed instantaneously. Hence, when considering the execution of such an operation, its starting time and its end time are instead considered.

The number that appears in the following definitions corresponds to a line number (i.e., to a register operation). Moreover, "*b*" stands for "beginning" while "*e*" stands for "end".

1. $\tau_e^x(1)$ is the time instant at which p_x terminates the assignment $FLAG[x] \leftarrow up$ (line 1).
2. $\tau_e^x(2)$ is the time instant at which p_x terminates the execution of line 2. Hence, at time $\tau_e^x(2)$ the non-atomic register $MY\_TURN[x]$ contains the value used by p_x to enter the critical section.
3. $\tau_b^x(3)$ is the time instant at which p_x starts the execution of line 3. This means that a process that reads $FLAG[x]$ during the time interval $[\tau_e^x(1)..\tau_b^x(3)]$ necessarily obtains the value up.
4. $\tau_b^x(5, y)$ is the time instant at which p_x starts its last evaluation of the waiting predicate (with respect to $FLAG[y]$) at line 5. This means that p_x has obtained the value $down$ from $FLAG[y]$.
5. Let us notice that, as it is the only process which writes into $MY\_TURN[x]$, p_x can save its value in a local variable. This means that the reading of $MY\_TURN[x]$ entails no access to the shared memory. Moreover, as far as a register $MY\_TURN[y]$ $(y \neq x)$ is concerned, we consider that p_x reads it once each time it evaluates the predicate of line 6.

   $\tau_b^x(6, y)$ is the time instant at which p_x starts its last reading of $MY\_TURN[y]$. Hence, the value $turn$ it reads from $MY\_TURN[y]$ is such that $(turn = 0) \vee \langle MY\_TURN[x], x\rangle < \langle turn, y\rangle$.

Terminology Let us remember that a process p_x is "in the doorway" when it executes line 2. We also say that it "is in the bakery" when it executes lines 4–9. Hence, when it is in the bakery, p_x is in the waiting room, inside the critical section, or executing release_mutex(x).

Lemma 1 *Let p_i and p_j be two processes that are in the bakery and such that p_i entered the bakery before p_j enters the doorway. Then $MY\_TURN[i] < MY\_TURN[j]$.*

Proof Let $turn_i$ be the value used by p_i at line 6. As p_i is in the bakery (i.e., executing lines 4–9) before p_j enters the doorway (line 2), it follows that $MY\_TURN[i]$ was assigned the value $turn_i$ before p_j reads it at line 2. Hence, when p_j reads the safe register $MY\_TURN[i]$, there is no concurrent write and p_j consequently obtains the value $turn_i$. It follows that the value $turn_j$ assigned by p_j to $MY\_TURN[j]$ is such that $turn_j \geq turn_i + 1$, from which the lemma follows. □

Lemma 2 *Let p_i and p_j be two processes such that p_i is inside the critical section while p_j is in the bakery. Then $\langle MY\_TURN[i], i\rangle < \langle MY\_TURN[j], j\rangle$.*

Proof Let us notice that, as p_j is inside the bakery, it can be inside the critical section.

As process p_i is inside the critical section, it has read $down$ from $FLAG[j]$ at line 5 (and exited the corresponding **wait** statement). It follows that, according to the timing of this read of $FLAG[j]$ that returned the value $down$ to p_i and the updates of $FLAG[j]$ by p_j to up at line 1 or $down$ at line 3 (the only lines where $FLAG[j]$ is modified), there are two cases to consider (Fig. 2.26).

As p_i reads $down$ from $FLAG[j]$, we have either $\tau_b^i(5, j) < \tau_e^j(1)$ or $\tau_e^i(5, j) > \tau_b^j(3)$ (see Fig. 2.26). This is because if we had $\tau_b^i(5, j) > \tau_e^j(1)$, p_i would necessarily have read up from $FLAG[j]$ (left part of the figure), and, if we had $\tau_b^i(5, j) < \tau_b^j(3)$, p_i would necessarily have also read up from $FLAG[j]$ (right part of the figure). Let us consider each case:

- Case 1: $\tau_b^i(5, j) < \tau_e^j(1)$ (left part of Fig. 2.26). In this case process, p_i has entered the bakery before process p_j enters the doorway. It then follows from Lemma 1 that $MY\_TURN[i] < MY\_TURN[j]$, which proves the lemma for this case.
- Case 2: $\tau_e^i(5, j) > \tau_b^j(3)$ (right part of Fig. 2.26). As p_j is sequential, we have $\tau_e^j(2) < \tau_b^j(3)$ (P1). Similarly, as p_i is sequential, we also have $\tau_b^i(5, j) < \tau_b^i(6, j)$ (P2). Combing (P1), (P2), and the case assumption, namely $\tau_b^j(3) < \tau_b^i(5, j)$, we obtain
$$\tau_e^j(2) < \tau_b^j(3) < \tau_e^i(5, j) < \tau_b^i(6, j);$$

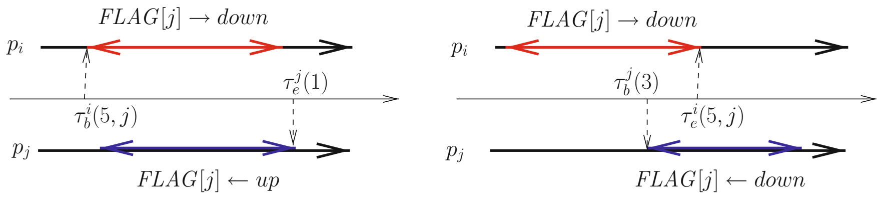


Fig. 2.26 The two cases where p_j updates the safe register $FLAG[j]$

i.e., $\tau_e^j(2) < \tau_b^i(6, j)$ from which we conclude that the last read of $MY\_TURN[j]$ by p_i occurred after the safe register $MY\_TURN[j]$ obtained its value (say $turn_j$).
As p_i is inside the critical section (lemma assumption), it exited the second **wait** statement because $(MY\_TURN[j] = 0) \vee \langle MY\_TURN[i], i\rangle < \langle MY\_TURN[j], j\rangle$. Moreover, as p_j was in the bakery before p_i executed line 6 ($\tau_e^j(2) < \tau_b^i(6, j)$), we have $MY\_TURN[j] = turn_j \neq 0$. It follows that we have $\langle MY\_TURN[i], i\rangle < \langle MY\_TURN[j], j\rangle$, which terminates the proof of the lemma. □

Theorem 9 *Lamport's bakery algorithm satisfies mutual exclusion and the bounded bypass liveness property where $f(n) = n - 1$.*

Proof Proof of the mutual exclusion property. The proof is by contradiction. Let us assume that p_i and p_j $(i \neq j)$ are simultaneously inside the critical section. We have the following:

- As p_i is inside the critical section and p_j is inside the bakery, we can apply Lemma 2. We then obtain: $\langle MY\_TURN[i], i\rangle < \langle MY\_TURN[j], j\rangle$.
- Similarly, as p_j is inside the critical section and p_i is inside the bakery, applying Lemma 2, we obtain: $\langle MY\_TURN[j], j\rangle < \langle MY\_TURN[i], i\rangle$.

As $i \neq j$, the pairs $\langle MY\_TURN[j], j\rangle$ and $\langle MY\_TURN[i], i\rangle$ are totally ordered. It follows that each item contradicts the other, from which the mutex property follows.

Proof of the FIFO order liveness property. The proof shows first that the algorithm is deadlock-free. It then shows that the algorithm satisfies the bounded bypass property where $f(n) = n - 1$ (i.e., the FIFO order as defined on the pairs $\langle MY\_TURN[x], x\rangle$).

The proof that the algorithm is deadlock-free is by contradiction. Let us assume that processes have invoked acquire_mutex() and no process exits the waiting room (lines 4–7). Let Q be this set of processes. (Let us notice that, for any other process p_j, we have $FLAG[j] = down$ and $MY\_TURN[j] = 0$.) As the number of processes is bounded and no process has to wait in the doorway, there is a time after which we have $\forall\, j \in \{1, \ldots, n\} : FLAG[j] = down$, from which we conclude that no process of Q can be blocked forever in the **wait** statement of line 5.

By construction, the pairs $\langle MY\_TURN[x], x\rangle$ of the processes $p_x \in Q$ are totally ordered. Let $\langle MY\_TURN[i], i\rangle$ be the smallest one. It follows that, eventually, when evaluated by p_i, the predicate associated with the **wait** statement of line 6 is satisfied for any j. Process p_i then enters the critical section, which contradicts the deadlock assumption and proves that the algorithm is deadlock-free.

To show the FIFO order liveness property, let us consider a pair of processes p_i and p_j that are competing for the critical section and such that p_j wins and after exiting the critical section it invokes acquire_mutex(j) again, executes its doorway, and enters the bakery. Moreover, let us assume that p_i is still waiting to enter the critical section. Let us observe that we are then in the context defined in Lemma 1: p_i and p_j are in the bakery and p_i entered the bakery before p_j enters the doorway.

We then have $MY\_TURN[i] < MY\_TURN[j]$, from which we conclude that p_j cannot bypass again p_i. As there are n processes, in the worst case p_i is competing with all other processes. Due to the previous observation and the fact that there is no deadlock, it can lose at most $n - 1$ competitions (one with respect to each other process p_j (which enters the critical section before p_i), which proves the bounded bypass liveness property with $f(n) = n - 1$. □

2.3.3 A Bounded Mutex Algorithm

This section presents a second mutex algorithm which does not require underlying atomic registers. This algorithm is due to A. Aravind (2011). Its design principles are different from the ones of the bakery algorithm.

Principle of the algorithm The idea that underlies the design of this algorithm is to associate a date with each request issued by a process and favor the competing process which has the oldest (smallest) request date. To that end, the algorithm ensures that (a) the dates associated with requests are increasing and (b) no two process requests have the same date.

More precisely, let us consider a process p_i that exits the critical section. The date of its next request (if any) is computed in advance when, just after p_i has used the critical section, it executes the corresponding release_mutex() operation. In that way, the date of the next request of a process is computed while this process is still "inside the critical section". As a consequence, the sequence of dates associated with the requests is an increasing sequence of consecutive integers and no two requests (from the same process or different processes) are associated with the same date.

From a liveness point of view, the algorithm can be seen as ensuring a *least recently used* (LRU) priority: the competing process whose previous access to the critical section is the oldest (with respect to request dates) is given priority when it wants to enter the critical section.

Safe registers associated with each process The following three SWMR safe registers are associated with each process p_i:

- $FLAG[i]$, whose domain is $\{down, up\}$. It is initialized to up when p_i wants to enter the critical section and reset to $down$ when p_i exits the critical section.
- If p_i is not competing for the critical section, the safe register $DATE[i]$ contains the (logical) date of its next request to enter the critical section. Otherwise, it contains the logical date of its current request.

  $DATE[i]$ is initialized to i. Hence, no two processes start with the same date for their first request. As already indicated, p_i will compute its next date (the value that will be associated with its next request for the critical section) when it exits the critical section.
- $STAGE[i]$ is a binary control variable whose domain is $\{0, 1\}$. Initialized to 0, it is set to 1 by p_i when p_i sees $DATE[i]$ as being the smallest date among the

```
operation acquire_mutex($i$) is
 (1)  $FLAG[i] \leftarrow up$;
 (2)  repeat $STAGE[i] \leftarrow 0$;
 (3)          wait $(\forall\, j \neq i : (FLAG[j] = down) \vee (DATE[i] < DATE[j]))$;
 (4)          $STAGE[i] \leftarrow 1$
 (5)  until $\forall\, j \neq i : (STAGE[j] = 0)$ end repeat;
 (6)  return()
end operation.

operation release_mutex($i$) is
 (7)  $DATE[i] \leftarrow \max(DATE[1], ..., DATE[n]) + 1$;
 (8)  $STAGE[i] \leftarrow 0$;
 (9)  $FLAG[i] \leftarrow down$;
 (10) return()
end operation.
```

Fig. 2.27 Aravind's mutual exclusion algorithm

dates currently associated with the processes that it perceives as competing for the critical section. The sequence of successive values taken by *STAGE*$[i]$ (including its initial value) is defined by the regular expression $0((0, 1)^+, 0)^*$.

Description of the algorithm Aravind's algorithm is described in Fig. 2.27. When a process p_i invokes acquire_mutex(i) it first sets its flag *FLAG*$[i]$ to *up* (line 1), thereby indicating that it is interested in the critical section. Then, it enters a loop (lines 2–5), at the end of which it will enter the critical section. The loop body is made up of two stages, denoted 0 and 1. Process p_i first sets *STAGE*$[i]$ to 0 (line 2) and waits until the dates of the requests of all the processes that (from its point of view) are competing for the critical section are greater than the date of its own request. This is captured by the predicate $(\forall\, j \neq i : (FLAG[j] = down) \vee (DATE[i] < DATE[j]))$, which is asynchronously evaluated by p_i at line 3. When, this predicate becomes true, p_i proceeds to the second stage by setting *STAGE*$[i]$ to 1 (line 1).

Unfortunately, having the smallest request date (as asynchronously checked at line 3 by a process p_i) is not sufficient to ensure the mutual exclusion property. More precisely, several processes can simultaneously be at the second stage. As an example let us consider an execution in which p_i and p_j are the only processes that invoke acquire_mutex() and are such that $DATE[i] = a < DATE[j] = b$. Moreover, p_j executes acquire_mutex() before p_i does. As all flags (except the one of p_j) are equal to *down*, p_j proceeds to stage 1 and, being alone in stage 1, exits the loop and enters the critical section. Then, p_i executes acquire_mutex(). As $a < b$, p_i does not wait at line 3 and is allowed to proceed to the second stage (line 4). This observation motivates the predicate that controls the end of the **repeat** loop (line 5). More precisely, a process p_i is granted the critical section only if it is the only process which is at the second stage (as captured by the predicate $\forall\, j \neq i : (STAGE[j] = 0)$ evaluated by p_i at line 5).

Finally, when a process p_i invokes release_mutex(i), it resets its control registers $STAGE[i]$ and $FLAG[i]$ to their initial values (0 and *down*, respectively). Before these updates, p_i benefits from the fact that it is still "inside the critical section" to compute the date of its next request and save it in $DATE[i]$ (line 7). It is important to see that no process p_j modifies $DATE[j]$ while p_i reads the array $DATE[1..n]$. Consequently, despite the fact that the registers are only SWMR safe registers (and not atomic registers), the read of any $DATE[j]$ at line 7 returns its exact value. Moreover, it also follows from this observation that no two requests have the same date and the sequence of dates used by the algorithm is the sequence of natural integers.

Theorem 10 *Aravind's algorithm (described in* Fig. 2.27*) satisfies mutual exclusion and the bounded bypass liveness property where* $f(n) = n - 1$.

Proof The proof of the mutual exclusion property is by contradiction. Let us assume that both p_i and p_j ($i \neq j$) are in the critical section.

Let $\tau_b^i(4)$ (or $\tau_e^i(4)$) be the time instant at which p_i starts (or terminates) writing $STAGE[i]$ at line 4 and $\tau_b^i(5, j)$ (or $\tau_e^i(5, j)$) be the time instant at which p_i starts (or terminates) reading $STAGE[j]$ for the last time at line 5 (before entering the critical section). These time instants are depicted in Fig. 2.28. By exchanging i and j we obtain similar notations for time instants associated with p_j.

As p_i is inside the critical section, it has read 0 from $STAGE[j]$ at line 5 and consequently we have $\tau_b^i(5, j) < \tau_e^j(4)$ (otherwise, p_i would necessarily have read 1 from $STAGE[j]$). Moreover, as p_i is sequential we have $\tau_e^i(4) < \tau_b^i(5, j)$, and as p_j is sequential, we have $\tau_e^j(4) < \tau_b^j(5, i)$. Piecing together the inequalities, we obtain

$$\tau_e^i(4) < \tau_b^i(5, j) < \tau_e^j(4) < \tau_b^j(5, i),$$

from which we conclude $\tau_e^i(4) < \tau_b^j(5, i)$, i.e., the last read of $STAGE[i]$ by p_j at line 5 started after p_i had written 1 into it. Hence, the last read of $STAGE[i]$ by p_j returned 1 which contradicts the fact that it is inside the critical section simultaneously with p_i. (A similar reasoning shows that, if p_j is inside the critical section, p_i cannot be.)

Before proving the liveness property, let us notice that at most one process at a time can modify the array $DATE[1..n]$. This follows from the fact that the algorithm satisfies the mutual exclusion property (proved above) and line 7 is executed by a process p_i before it resets $STAGE[i]$ to 0 (at line 8), which is necessary to allow

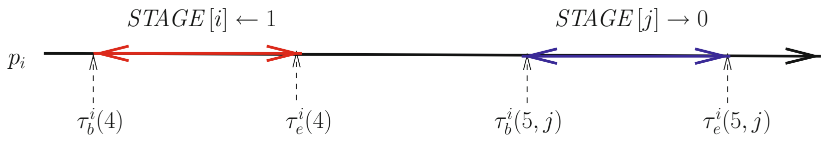


Fig. 2.28 Relevant time instants in Aravind's algorithm

another process p_j to enter the critical section (as the predicate of line 5 has to be true when evaluated by p_j). It follows from the initialization of the array $DATE[1..n]$ and the previous reasoning that no two requests can have the same date and the sequence of dates computed in mutual exclusion at line 7 by the processes is the sequence of natural integers (Observation OB).

As in the proof of Lamport's algorithm, let us first prove that there is no deadlock. Let us assume (by contradiction) that there is a non-empty set of processes Q that have invoked acquire_mutex() and no process succeeds in entering the critical section. Let p_i be the process of Q with the smallest date. Due to observation OB, there is a single process p_i. It then follows that, after some finite time, p_i is the only process whose predicate at line 3 is satisfied. Hence, after some time, p_i is the only process such that $STAGE[i] = 1$, which allows it to enter the critical section. This contradicts the initial assumption and proves the deadlock-freedom property.

As a single process at a time can modify its entry of the array $DATE$, it follows that a process p_j that exits the critical section updates its register $DATE[j]$ to a value greater than all the values currently kept in $DATE[1..n]$. Consequently, after p_j has executed line 7, all the other processes p_i which are currently competing for the critical section are such that $DATE[i] < DATE[j]$. Hence, as we now have $(FLAG[i] = up) \wedge (DATE[i] < DATE[j])$, the next request (if any) issued by p_j cannot bypass the current request of p_i, from which the starvation-freedom property follows.

Moreover, it also follows from the previous reasoning that, if p_i and p_j are competing and p_j wins, then as soon as p_j has exited the critical section p_i has priority over p_j and can no longer be bypassed by it. This is nothing else than the bounded bypass property with $f(n) = n - 1$ (which defines a FIFO order property). $\square$

Bounded mutex algorithm Each safe register $MY\_TURN[i]$ of Lamport's algorithm and each safe register $DATE[i]$ of Aravind's algorithm can take arbitrary large values. It is shown in the following how a simple modification of Aravind's algorithm allows for bounded dates. This modification relies on the notion of an MWMR safe register.

MWMR safe register An *MWMR safe* register is a safe register that can be written and read by several processes. When the write operations are sequential, an MWMR safe register behaves as an SWMR safe register. When write operations are concurrent, the value written into the register is any value of its domain (not necessarily a value of a concurrent write).

Said differently, to be meaningful, an algorithm based on MWMR safe registers has to prevent write operations on an MWMR safe register from being concurrent in order for the write operations to be always meaningful. The behavior of an MWMR safe register is then similar to the behavior of an SWMR safe register in which the "single writer" is implemented by several processes that never write at the same time.

From unbounded dates to bounded dates Let us now consider that each safe register $DATE[i]$, $1 \leq i \leq n$, is an MWMR safe register: any process p_i can write any register $DATE[j]$. MWMR safe registers allow for the design of a (particularly simple) bounded mutex algorithm. The domain of each register $DATE[j]$ is now $[1..N]$ where $N \geq 2n$. Hence, all registers are safe and have a bounded domain. In the following we consider $N = 2n$. A single bit is needed for each safe register $FLAG[j]$ and each safe register $STAGE[j]$, and only $\lceil \log_2 N \rceil$ bits are needed for each safe register $DATE[j]$.

In a very interesting way, no statement has to be modified to obtain a bounded version of the algorithm. A single new statement has to be added, namely the insertion of the following line 7′ between line 7 and line 8:

(7′) **if** $(DATE[i] \geq N)$ **then for all** $j \in [1..n]$ **do** $DATE[j] \leftarrow j$ **end for end if**.

This means that, when a process p_i exiting the critical section updates its register $DATE[i]$ and this update is such that $DATE[i] \geq N$, p_i resets all date registers to their initial values. As for line 7, this new line is executed before $STAGE[i]$ is reset to 0 (line 8), from which it follows that it is executed in mutual exclusion and consequently no two processes can concurrently write the same MWMR safe register $DATE[j]$. Hence, the MWMR safe registers are meaningful.

Moreover, it is easy to see that the date resetting mechanism is such that each date d, $1 \leq d \leq n$, is used only by process p_d, while each date d, $n+1 \leq d \leq 2n$ can be used by any process. Hence, $\forall d \in \{1, \ldots, n\}$ we have $DATE[d] \in \{d, n+1, n+2, \ldots, 2n\}$.

Theorem 11 *When considering Aravind's mutual exclusion algorithm enriched with line 7′ with $N \geq 2n$, a process encounters at most one reset of the array $DATE[1..n]$ while it is executing* acquire_mutex().

Proof Let p_i be a process that executes acquire_mutex() while a reset of the array $DATE[1..n]$ occurs. If p_i is the next process to enter the critical section, the theorem follows. Otherwise, let p_j be the next process which enters the critical section. When p_j exits the critical section, $DATE[j]$ is updated to $\max(DATE[1], \ldots, DATE[n]) + 1 = n + 1$. We then have $FLAG[i] = up$ and $DATE[i] < DATE[j]$. It follows that, if there is no new reset, p_j cannot enter again the critical section before p_i.

In the worst case, after the reset, all the other processes are competing with p_i and p_i is p_n (hence, $DATE[i] = n$, the greatest date value after a reset). Due to line 3 and the previous observation, each other process p_j enters the critical section before p_i and $\max(DATE[1], \ldots, DATE[n])$ becomes equal to $n + (n-1)$. As $2n - 1 < 2n \leq N$, none of these processes issues a reset. It follows that p_i enters the critical section before the next reset. (Let us notice that, after the reset, the invocation issued by p_i can be bypassed only by invocations (pending invocations issued before the reset or new invocations issued after the reset) which have been issued by processes p_j such that $j < i$). □

The following corollary is an immediate consequence of the previous theorem.

Corollary 2 *Let $N \geq 2n$. Aravind's mutual exclusion algorithm enriched with line 7′ satisfies the starvation-freedom property.*

(Different progress conditions that this algorithm can ensure are investigated in Exercise 6.)

Bounding the domain of the safe registers has a price. More precisely, the addition of line $7'$ has an impact on the maximal number of bypasses which can now increase up to $f(n) = 2n-2$. This is because, in the worst case where all the processes always compete for the critical section, before it is allowed to access the critical section, a process can be bypassed $(n-1)$ times just before a reset of the array *DATE* and, due to the new values of $DATE[1..n]$, it can again be bypassed $(n-1)$ times just after the reset.

2.4 Summary

This chapter has presented three families of algorithms that solve the mutual exclusion problem. These algorithms differ in the properties of the base operations they rely on to solve mutual exclusion.

Mutual exclusion is one way to implement atomic objects. Interestingly, it was shown that implementing atomicity does not require the underlying read and write operations to be atomic.

2.5 Bibliographic Notes

- The reader will find surveys on mutex algorithms in [24, 231, 262]. Mutex algorithms are also described in [41, 146].
- Peterson's algorithm for two processes and its generalization to n processes are presented in [224].

  The first tournament-based mutex algorithm is due to G.L. Peterson and M.J. Fischer [227].

  A variant of Peterson's algorithm in which all atomic registers are SWMR registers due to J.L.W. Kessels is presented in [175].
- The contention-abortable mutex algorithm is inspired from Lamport's fast mutex algorithm [191]. Fischer's synchronous algorithm is described in [191].

  Lamport's fast mutex algorithm gave rise to the splitter object as defined in [209].

  The notion of fast algorithms has given rise to the notion of *adaptive* algorithms (algorithms whose cost is related to the number of participating processes) [34].
- The general construction from deadlock-freedom to starvation-freedom that was presented in Sect. 2.2.2 is from [262]. It is due to Y. Bar-David.

- The notions of safe, regular, and atomic read/write registers are due to L. Lamport. They are presented and investigated in [188, 189]. The first intuition on these types of registers appears in [184].

  It is important to insist on the fact that "non-atomic" does not mean "arbiter-free". As defined in [193], "An arbiter is a device that makes a discrete decision based on a continuous range of values". Binary arbiters are the most popular. Actually, the implementation of a safe register requires an arbiter. The notion of arbitration-free synchronization is discussed in [193].
- Lamport's bakery algorithm is from [183], while Aravind's algorithm and its bounded version are from [28].
- A methodology based on model-checking for automatic discovery of mutual exclusion algorithms has been proposed by Y. Bar-David and G. Taubenfeld [46]. Interestingly enough, this methodology is both simple and computationally feasible. New algorithms obtained in this way are presented in [46, 262].
- Techniques (and corresponding algorithms) suited to the design of locks for NUMA and CC-NUMA architectures are described in [86, 200]. These techniques take into account non-uniform memories and caching hierarchies.
- A *combiner* is a thread which, using a coarse-grain lock, serves (in addition to its own synchronization request) active requests announced by other threads while they are waiting by performing some form of spinning. Two implementations of such a technique are described in [173]. The first addresses systems that support coherent caches, whereas the second works better in cacheless NUMA architectures.

2.6 Exercises and Problems

1. Peterson's algorithm for two processes uses an atomic register denoted *TURN* that is written and read by both processes. Design a two-process mutual exclusion algorithm (similar to Peterson's algorithm) in which the register *TURN* is replaced by two SWMR atomic registers $TURN[i]$(which can be written only by p_i) and $TURN[j]$(which can be written only by p_j). The algorithm will be described for p_i where $i \in \{0, 1\}$ and $j = (i + 1) \mod 2$.

   Solution in [175].
2. Considering the tournament-based mutex algorithm, show that if the base two-process mutex algorithm is deadlock-free then the n-process algorithm is deadlock-free.

3. Design a mutex starvation-free algorithm whose cost (measured by the number of shared memory accesses) depends on the number of processes which are currently competing for the critical section. (Such an algorithm is called *adaptive*.)

   Solutions in [23, 204, 261].

4. Design a fast deadlock-free mutex synchronous algorithm. "Fast" means here that, when no other process is interested in the critical section when a process p requires it, then process p does not have to execute the delay() statement.

   Solution in [262].

5. Assuming that all registers are atomic (instead of safe), modify Lamport's bakery algorithm in order to obtain a version in which all registers have a bounded domain.

   Solutions in [171, 261].

6. Considering Aravind's algorithm described in Fig. 2.27 enriched with the reset line (line $7'$):

   - Show that the safety property is independent of N; i.e., whatever the value of N (e.g., $N = 1$), the enriched algorithm allows at most one process at a time to enter the critical section.
   - Let $x \in \{1, \ldots, n-1\}$. Which type of liveness property is satisfied when $N = x + n$ (where n is the number of processes).
   - Let $I = \{i_1, \ldots, i_z\} \subseteq \{1, \ldots, n\}$ be a predefined subset of process indexes. Modify Aravind's algorithm in such a way that starvation-freedom is guaranteed only for the processes p_x such that $x \in I$. (Let us notice that this modification realizes a type of priority for the processes whose index belong to I in the sense that the algorithm provides now processes with two types of progress condition: the invocations of acquire_mutex() issued by any process p_x with $x \in I$ are guaranteed to terminate, while they are not if $x \notin I$.) Modify Aravind's algorithm so that the set I can be dynamically updated (the main issue is the definition of the place where such a modification has to introduced).

Chapter 3
Lock-Based Concurrent Objects

After having introduced the notion of a concurrent object, this chapter presents lock-based methodologies to implement such objects. The first one is based on a low-level synchronization object called a semaphore. The other ones are based on linguistic constructs. One of these constructs is based on an imperative approach (monitor construct), while the other one is based on a declarative approach (path expression construct). This chapter closes the first part of the book devoted to lock-based synchronization.

Keywords Declarative synchronization · Imperative synchronization · Lock-based implementation · Monitor · Path expression · Predicate transfer · Semaphore

3.1 Concurrent Objects

3.1.1 Concurrent Object

Definition An object type is defined by a finite set of operations and a specification describing the correct behaviors of the objects of that type. The internal representation of an object is hidden to the processes (and several objects of the same type can have different implementations). The only way for a process to access an object of a given type is by invoking one of its operations.

A *concurrent* object is an object that can be accessed concurrently by several processes. The specification of such an object can be sequential or not. "Sequential" means that all correct behaviors of the object can be described with sequences (traces). Not all concurrent objects have a sequential specification.

Example As an example, let us consider the classical unbounded stack object type. Any object of such a type provides processes with a push() operation and a pop() operation. As the stack is unbounded, the push() operation can always be executed. As far as the pop() operation is concerned, let us assume that it returns the default value ⊥ when the stack is empty. Hence, both operations can always be executed whatever the current state of the stack (such operations are said to be *total*). The sequential specification of such a stack is the set of all the sequences of push() and pop() operations that satisfy the "last in, first out" (LIFO) property ("last in" being ⊥ when the stack is empty). Differently, as indicated in the first chapter, a rendezvous object has no sequential specification.

3.1.2 Lock-Based Implementation

A simple way to implement a concurrent object defined by a sequential specification consists in using a lock to force each invocation of an operation to execute in mutual exclusion. In that way, a single process at a time can access the internal representation of the object. It follows that sequential algorithms can be used to implement the object operations.

As an example, let us consider a sequential stack *S_STACK* (i.e., a stack which was designed to be used by a sequential program). As previously noticed, its internal representation and the code of the sequential algorithms implementing its push() and pop() operations are hidden to its user program (such a description is usually kept in a library).

Let conc_push() and conc_pop() be the operations associated with a concurrent stack denoted *C_STACK*. A simple way to obtain an implementation of *C_STACK* is as follows:

- Its internal representation consists of an instance of a sequential stack *S_STACK* plus a lock instance *LOCK* as depicted in Fig. 3.1;
- The algorithms implementing its conc_push() and conc_pop() operations are based on this internal representation as described in Fig. 3.2.

This methodology has the great advantage of clearly separating the control part (the underlying *LOCK* object) from the data part (the underlying *S_SACK* object).

Let us notice that each instance of a concurrent stack has its own internal representation, i.e., is made up of its own instances of both a lock and a sequential stack. This means that, if we have two concurrent stacks C_STACK_1 and C_STACK_2, the

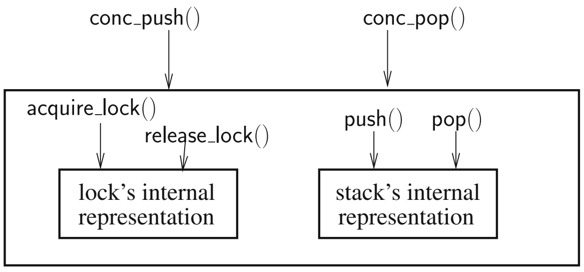

conc_push()
conc_pop()
acquire_lock()
release_lock()
push()
pop()
lock's internal representation
stack's internal representation

Fig. 3.1 From a sequential stack to a concurrent stack: structural view

```
operation $C\_STACK$.conc_push($i, v$) is
    $LOCK$.acquire_lock($i$); $S\_STACK$.push($v$); $LOCK$.release_lock($i$);
    return()
end operation.

operation $C\_STACK$.conc_pop($i$) is
    $LOCK$.acquire_lock($i$); $r \leftarrow S\_STACK$.pop(); $LOCK$.release_lock($i$);
    return($r$)
end operation.
```

Fig. 3.2 From a sequential to a concurrent stack (code for p_i)

first uses a sequential stack S_STACK_1 and a lock instance $LOCK_1$, while the second uses another sequential stack S_STACK_2 and another lock instance $LOCK_2$. Hence, as $LOCK_1$ and $LOCK_2$ are distinct locks, the operations on C_STACK_1 and C_STACK_2 are not prevented from being concurrent.

3.2 A Base Synchronization Object: the Semaphore

3.2.1 The Concept of a Semaphore

Definition of a semaphore A semaphore S is a shared counter which can be accessed by two atomic operations denoted S.down() and S.up(). The specification of a semaphore is defined as follows:

- A semaphore S is initialized to a non-negative value s_0.
- The predicate $S \geq 0$ is always satisfied (i.e., the counter never becomes negative).
- S.down() atomically decreases S by 1.
- S.up() atomically increases S by 1.

It is easy to see that the operation S.up() can always be executed. Differently, the operation S.down() can be executed only if its execution does not entail the violation of the invariant $S \geq 0$. When $S = 1$ and two or more processes invoke S.down(), one of them succeeds (and the semaphore becomes equal to 0) while the other becomes blocked. One of them will be unblocked when a process executes S.up(), etc.

Invariant Let #(S.down) (or #(S.up)) be the number of invocations of S.down() (or S.up()) that have terminated. It follows from the definition of a semaphore S that the following relation is always satisfied:

$$S = s_0 + \#(S.\mathsf{up}) - \#(S.\mathsf{down}).$$

Locks, tokens, and semaphores A semaphore can be seen as a generalization of a lock; namely, a lock is a semaphore S which is initialized to 1 and where S.down() and S.up() are renamed acquire_lock() and release_lock(), respectively.

More generally, a semaphore can be seen as a token manager. Initially, there are s_0 tokens. Then, each invocation of S.down() consumes one token while each invocation of S.up() generates one token. If a process invokes S.down() while there are no tokens, it has to wait until a token is created in order to consume it. The value of S defines the actual number of tokens which can be used by the processes.

Semaphore variants A semaphore S is *strong* if the processes that are blocked are unblocked in the order in which they became blocked. Otherwise, the semaphore is *weak*.

A *binary* semaphore S is a semaphore that can take only the values 0 and 1. Hence, an invocation of S.down() blocks the invoking process if $S = 0$, and an invocation of S.up() blocks it if $S = 1$.

A semaphore S is *private* to a process p_i if only p_i can invoke S.down(). The other processes can only invoke S.up() to send "signals" to p_i. (This is analogous to the notion of an SWMR atomic registers that can be written by a single process.)

Implementation of a semaphore Semaphores were initially introduced to help solve both synchronization and scheduling problems. More precisely, they were introduced to prevent *busy* waiting (waiting loops as used in the previous chapter devoted to mutual exclusion) in systems where there are more processes than processors.

To that end a semaphore S is implemented by two underlying objects:

- A counter, denoted $S.count$ and initialized to s_0. As we will see, this counter can become negative and is not to be confused with the value of S, which is never negative.
- A queue, denoted $S.queue$, which is initially empty. (A FIFO queue gives rise to a strong semaphore.)

A schematic implementation of the operations S.down() and S.up() is described in Fig. 3.3. (This code is mainly designed to assign processors to processes.) It is assumed that, for each semaphore S, each operation is executed in mutual exclusion.

Let $nb\_blocked(S)$ denote the number of processes currently blocked during their invocations of S.down(). The reader can verify that the following relation is a invariant of the previous implementation:

$$\textbf{if } (S.count \geq 0) \textbf{ then } nb\_blocked(S) = 0 \textbf{ else } nb\_blocked(S) = |S.count| \textbf{ end if}.$$

Hence, when it is negative, the implementation counter $S.count$ provides us with the number of processes currently blocked on the semaphore S. Differently, when it is non-negative, the value of $S.count$ is the value of the semaphore S.

```
operation $S$.down() is
    $S.count \leftarrow S.count - 1$;
    if ($S.count < 0$) then
        the invoking process is blocked and added to $S.queue$; the control is given to the scheduler
    end if;
    return()
end operation.

operation $S$.up() is
    $S.count \leftarrow S.count + 1$;
    if ($S.count \leq 0$) then
        remove the first process in $S.queue$ which can now be assigned a processor
    end if;
    return()
end operation.
```

Fig. 3.3 Implementing a semaphore(code for p_i)

3.2.2 Using Semaphores to Solve the Producer-Consumer Problem

The producer-consumer problem This problem was introduced in Chap. 1. We have to construct a concurrent object (usually called a *buffer*) defined by two operations denoted produce() and consume() such that produce() allows a process to deposit a new item in the buffer while consume() allows a process to withdraw an item from the buffer. The capacity of the buffer is $k > 0$ items. Moreover, the xth item that was produced has to be consumed exactly once and before the $(x + 1)$th item (see Fig. 1.4). It is (of course) assumed that the operation consume() is invoked enough times so that each item is consumed.

The Case of a Single Producer and a Single Consumer

Implementation data structures Considering the case where there is a single producer process and a single consumer process, let us construct a buffer, denoted B, whose size is $k \geq 1$. A simple semaphore-based implementation of a buffer is as follows. The internal representation is made up of two parts.

- Data part. This part is the internal representation of the buffer. It comprises three objects:
  - $BUF[0..(k-1)]$ is an array of read/write registers which are not required to be atomic or even safe. This is because the access to any register $BUF[x]$ is protected by the control part of the implementation that (as we are about to see) guarantees that the producer and the consumer cannot simultaneously access any $BUF[x]$.

The base read and write operations on $BUF[x]$ are denoted $BUF[x]$.read() and $BUF[x]$.write().

- *in* and *out* are two local variables containing array indexes whose domain is $[0..(k-1)]$; *in* is used by the producer to point to the next entry of *BUF* where an item can be deposited; *out* is used by the consumer to point to the next entry of *BUF* from which an item can be consumed. The law that governs the progress of these index variables is the addition $\mod k$, and we say that the buffer is *circular*.

- Control part. This part comprises the synchronization objects that allow the processes to never violate the buffer invariant. It consists of two semaphores:
  - The semaphore *FREE* counts the number of entries of the array *BUF* that can currently be used to deposit new items.This semaphore is initialized to k.
  - The semaphore *BUSY* counts the number of entries of the array *BUF* that currently contain items produced and not yet consumed. This semaphore is initialized to 0.

Production and consumption algorithms The algorithms implementing the buffer operations produce() and consume() are described in Fig. 3.4. When the producer invokes B.produce(*value*) (where *value* is the value of the item it wants to deposit into the buffer), it first checks if there is a free entry in the array *BUF*. The semaphore *FREE* is used to that end (line 1). When $FREE = 0$, the producer is blocked until an entry of the buffer is freed. Then, the producer deposits the next item value into $BUF[in]$ (line 2) and increases the index *in* so that it points to the next entry of the array. Finally, the producer signals that one more entry was produced (line 3).

The algorithm for the operation B.consume() is a control structure symmetric to that of the B.produce() operation, exchanging the semaphores *BUSY* and *FREE*.

```
operation $B$.produce($v$) is
 (1)  $FREE$.down();
 (2)  $BUF[in]$.write($v$); $in \leftarrow (in+1) \mod k$;
 (3)  $BUSY$.up();
 (4)  return()
end operation.

operation $B$.consume() is
 (5)  $BUSY$.down();
 (6)  $r \leftarrow BUF[out]$.read(); $out \leftarrow (out+1) \mod k$;
 (7)  $FREE$.up();
 (8)  return($r$)
end operation.
```

Fig. 3.4 A semaphore-based implementation of a buffer

When the consumer invokes B.consume(), it first checks if there is one entry in the array BUF that contains an item not yet consumed. The semaphore $BUSY$ is used to that end (line 5). When it is allowed to consume, the consumer consumes the next item value, i.e., the one kept in $BUF[out]$, saves it in a local variable r, and increases the index out (line 5). Finally, before returning that value saved in r (line 5), the consumer signals that one entry is freed; this is done by increasing the value of the semaphore $FREE$ (line 7).

Remark 1 It is important to repeat that, for any x, a register $BUF[x]$ is not required to satisfy special semantic requirements and that the value that is written (read) can be of any type and as large as we want (e.g., a big file). This is an immediate consequence of the fact each register $BUF[x]$ is accessed in mutual exclusion. This means that what is abstracted as a register $BUF[x]$ does not have to be constrained in any way. As an example, the operation $BUF[in]$.write(v) (line 2) can abstract several low-level write operations involving accesses to underlying disks which implement the register (and similarly for the operation $BUF[in]$.read() at line 2). Hence, the size of the items that are produced and consumed can be arbitrary large, and reading and writing them can take arbitrary (but finite) durations. This means that one can reasonably assume that the duration of the operations $BUF[in]$.write(v) and $BUF[in]$.read() (i.e., the operations which are in the data part of the algorithms) is usually several orders of magnitude greater than the execution of the rest of the algorithms (which is devoted to the control part).

Remark 2 It is easy to see that the values taken by the semaphores $FREE$ and $BUSY$ are such that $0 \leq FREE, BUSY \leq k$, but it is important to remark that a semaphore object does not offer an operation such as $FREE$.read() that would return the exact value of $FREE$. Actually, such an operation would be useless because there is no guarantee that the value returned by $FREE$.read() is still meaningful when the invoking process would use it ($FREE$ may have been modified by $FREE$.up() or $FREE$.down() just after its value was returned).

A semaphore S can be seen as an atomic register that could be modified by the operations fech&add() and fetch&sub() (which atomically add 1 and subtract 1, respectively) with the additional constraint that S can never become negative.

The case of a buffer with a single entry This is the case $k = 1$. Each of the semaphores $FREE$ and $BUSY$ takes then only the values 0 or 1. It is interesting to look at the way these values are modified. A corresponding cycle of production/consumption is depicted in Fig. 3.5.

Initially the buffer is empty, $FREE = 1$, and $BUSY = 0$. When the producer starts to deposit a value, the semaphore $FREE$ decreases from 1 to 0 and the buffer starts to being filled. When it has been filled, the producer raises $BUSY$ from 0 to 1. Hence, $FREE = 0 \wedge BUSY = 1$ means that a value has been deposited and can be consumed. When the consumer wants to read, it first decreases the semaphore $BUSY$ from 1 to 0 and then reads the value kept in the buffer. When, the reading is terminated, the consumer signals that the buffer is empty by increasing $FREE$ from

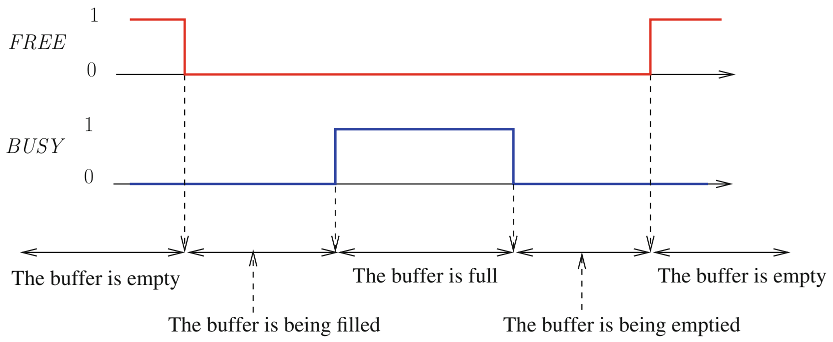


Fig. 3.5 A production/consumption cycle

0 to 1, and we are now in the initial configuration where $FREE = 1 \wedge BUSY = 0$ (which means that the buffer is empty).

When looking at the values of the semaphores, it is easy to see that we can never have both semaphores simultaneously equal to 1, from which we conclude that $\neg(FREE = 1 \wedge BUSY = 1)$ is an invariant relation that, for $k = 1$, characterizes the buffer implementation described in Fig. 3.5. More generally, when $k \geq 1$, the invariant is $0 \leq |FREE + BUSY| \leq k$.

The Case of Several Producers and Several Consumers

If there are several producers (consumers) the previous solution no longer works, because the control register *in* (*out*) now has to be an atomic register shared by all producers (consumers). Hence, the local variables *in* and *out* are replaced by the atomic registers *IN* and *OUT*. Moreover, (assuming $k > 1$) the read and update operations on each of these atomic registers have to be executed in mutual exclusion in order that no two producers simultaneously obtain the same value of *IN*, which could entail the write of an arbitrary value into $BUF[in]$. (And similarly for *out*.)

A simple way to solve this issue consists in adding two semaphores initialized to 1, denoted *MP* and *MC*. The semaphore *MP* is used by the producers to ensure that at most one process at a time is allowed to execute B.produce(); (similarly *MC* is used to ensure that no two consumers concurrently execute B.consume()). Albeit correct, such a solution can be very inefficient. Let us consider the case of a producer p_1 that is very slow while another producer p_2 is very rapid. If both p_1 and p_2 simultaneously invoke produce() and p_1 wins the competition, p_2 is forced to wait for a long time before being able to produce. Moreover, if there are several free entries in $BUF[0..(k-1)]$, it should be possible for p_1 and p_2 to write simultaneously in two different free entries of the array.

Additional control data To that end, in addition to the buffer $BUF[0..(k-1)]$ and the atomic registers *IN* and *OUT*, two arrays of atomic Boolean registers denoted

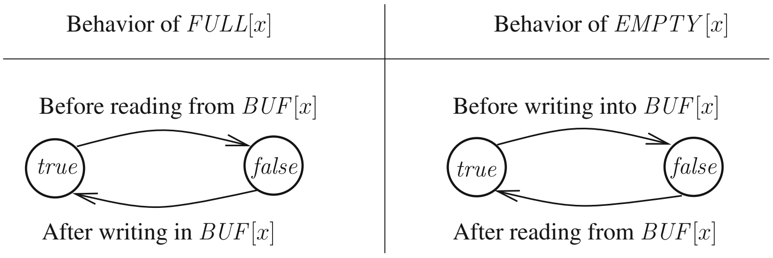

Behavior of $FULL[x]$
Behavior of $EMPTY[x]$
Before reading from $BUF[x]$
true
false
After writing in $BUF[x]$
Before writing into $BUF[x]$
true
false
After reading from $BUF[x]$

Fig. 3.6 Behavior of the flags $FULL[x]$ and $EMPTY[x]$

$FULL[0..(k-1)]$ and $EMPTY[0..(k-1)]$ are used. They are such that, for every x, the pair $\langle FULL[x], EMPTY[x]\rangle$ describes the current state of $BUF[x]$ (full, empty, being filled, being emptied). These registers have similar behaviors, one from the producer point of view and the other one from the consumer point of view. More precisely, we have the following:

- $FULL[x]$ (which is initialized to *false*) is set to *true* by a producer p just after it has written a new item value in $BUF[x]$. In that way, p informs the consumers that the value stored in $BUF[x]$ can be consumed. $FULL[x]$ is reset to *false* by a consumer c just after it has obtained the right to consume the item value kept in $BUF[x]$. In that way, c informs the other consumers that the value in $BUF[x]$ is not for them. To summarize: $FULL[x] \Leftrightarrow$ ($BUF[x]$ can be read by a consumer) (Fig. 3.6).
- $EMPTY[x]$ (which is initialized to *true*) is reset to *false* by a consumer c just after it has read the item value kept in $BUF[x]$. In that way, the consumer c informs the producers that $BUF[x]$ can be used again to deposit a new item value. $EMPTY[x]$ is set to *false* by a producer p just before it writes a new item value in $BUF[x]$. In that way, the producer p informs the other producers that $BUF[x]$ is reserved and they cannot write into it. To summarize: $EMPTY[x] \Leftrightarrow$ ($BUF[x]$ can be written by a producer).

The algorithm The code of the algorithms implementing produce() and consume() is described in Fig. 3.7. We describe only the code of the operation produce(). As in the base algorithms of Fig. 3.4, the code of the operation consume() is very similar.

When a producer p invokes produce(), it first executes $FREE$.down(), which blocks it until there is at least one free entry in $BUF[1..(k-1)]$ (line 1). Then, p_i executes MP.down(). As MP is initialized to 1 and used only at lines 2 and 7, this means that a single producer at a time can execute the control statements defined at lines 2–7. The aim of these lines is to give to p an index (my_index) such that no other producer is given the same index in order to write into $BUF[my\_index]$ (at line 8). To that end, the producer p scans (modulo k) the array $EMPTY[1..(k-1)]$ in order to find a free entry. Let us notice that there is necessarily such an entry because p has passed the statement $FREE$.down(). Moreover, so that each item that is written

```
operation $B$.produce($v$) is
(1)  $FREE$.down();
(2)  $MP$.down();
(3)     while $(\neg EMPTY[IN])$ do $IN \leftarrow (IN+1) \mod k$ end while;
(4)     $my\_index \leftarrow IN$;
(5)     $EMPTY[IN] \leftarrow false$;
(6)     $IN \leftarrow (IN+1) \mod k$;
(7)  $MP$.up();
(8)  $BUF[my\_index]$.write($v$);
(9)  $FULL[my\_index] \leftarrow true$;
(10) $BUSY$.up();
(11) return()
end operation.

operation $B$.consume() is
(12) $BUSY$.down();
(13) $MC$.down();
(14)    while $(\neg FULL[OUT])$ do $OUT \leftarrow (OUT+1) \mod k$ end while;
(15)    $my\_index \leftarrow OUT$;
(16)    $FULL[OUT] \leftarrow false$;
(17)    $OUT \leftarrow (OUT+1) \mod k$;
(18) $MC$.up();
(19) $r \leftarrow BUF[my\_index]$.read();
(20) $EMPTY[my\_index] \leftarrow true$;
(21) $FREE$.up();
(22) return($r$)
```

Fig. 3.7 An efficient semaphore-based implementation of a buffer

is consumed, a producer starts scanning at $EMPTY[IN]$ (and similarly a consumer starts scanning the array $FULL[1..(k-1)]$ at the entry $FULL[OUT]$). The first value of IN such that $EMPTY[IN]$ is true defines the value of *my_index* (lines 3–4). Then, p updates $EMPTY[IN]$ to *false* (line 5) and increases IN so that it points to the next entry that a producer will use to start its scan (line 6). The accesses to IN are then finished for p, and it releases the associated mutual exclusion (line 7).

Then, the producer p writes its item value into $BUF[my\_index]$ at line 8 (let us remember that this statement is the most expensive from a time duration point of view). When, the write is terminated, p informs the consumers that $BUF[my\_index]$ contains a new value; this is done by setting $FULL[my\_index]$ to *true* (line 9). Finally, (as in the base algorithm) the producer p indicates that one more value can be consumed by increasing the semaphore *BUSY* (line 10).

The reader can check that, for any x, when considering $BUF[x]$ the production/consumption cycle is exactly the same as the one described in Fig. 3.5 where the semaphore *FREE* and *BUSY* are replaced by the Boolean atomic registers $EMPTY[x]$ and $FULL[x]$, respectively. It follows that the invariant

$$\forall x \in [0..(k-1)] : (\neg EMPTY[x]) \vee (\neg FULL[x])$$

characterizes the buffer implementation given in Fig. 3.7. This invariant states that no $BUF[x]$ can be simultaneously full and empty. Other facts can be deduced from Fig. 3.7:

- $\neg EMPTY[x]) \wedge \neg FULL[x] \Rightarrow BUF[x]$ is currently being filled or emptied,
- $FULL[x] \Rightarrow BUF[x]$ contains a value not yet consumed, and
- $EMPTY[x] \Rightarrow BUF[x]$ does contain a value to be consumed.

3.2.3 Using Semaphores to Solve a Priority Scheduling Problem

Priority without preemption Let us consider a resource that has to be accessed in mutual exclusion. To that end, as we have seen before, we may build an object that provides the processes with two operations, denoted acquire() and release(), that are used to bracket any use of the resource.

Moreover, in addition to being mutually exclusive, the accesses to the resource have to respect a priority scheme defined as follows:

- Each request for the resource is given a priority $prio_i$ value by the invoking process p_i.
- When a process (which was granted the resource) releases it, the resource has to be given to the process whose pending request has the highest priority. (It is assumed that no two requests have the same priority. Process identities can be used for tie-breaking.)

Hence, when a process p_i wants to use the resource, it invokes the following operation:

```
operation use_resource($i, prio_i$) is
    acquire($i, prio_i$);
    access the resource;
    release($i$);
    return()
end operation.
```

Let us observe that this priority scheme is without preemption: when a process p has obtained the resource, it keeps it until it invokes release(), whatever the priority of the requests that have been issued by other processes after p was granted the resource.

Principle of the solution and base objects The object we are building can be seen as a room made up of two parts: a blackboard room where processes can post information to inform the other processes plus a sleeping room where processes go to wait when the resource is used by another process (see Fig. 3.8).

In order for the information on the blackboard to be consistent, at most one process at a time can access the room. To that end a semaphore denoted $MUTEX$ and initialized to 1 is used by the processes.

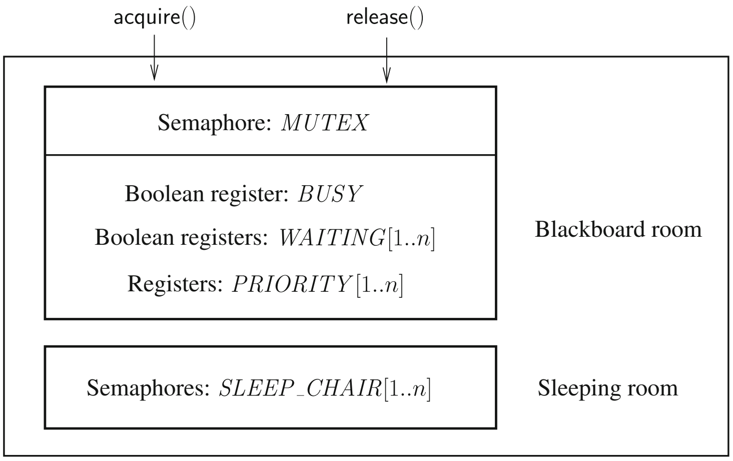

acquire()
release()
Semaphore: $MUTEX$
Boolean register: $BUSY$
Boolean registers: $WAITING[1..n]$
Registers: $PRIORITY[1..n]$
Blackboard room
Semaphores: $SLEEP\_CHAIR[1..n]$
Sleeping room

Fig. 3.8 Blackboard and sleeping rooms

The blackboard is made up of the following read/write registers. As the accesses to these objects are protected by the mutual exclusion semaphore *MUTEX*, they do not need to be atomic or even safe.

- A Boolean register denoted *BUSY* which is initialized to *false*. This Boolean is equal to *true* if and only if one process is allowed to use the resource.
- An array of Boolean registers *WAITING*$[1..n]$ initialized to $[\mathit{false},\ldots,\mathit{false}]$. *WAITING*$[i]$ is set to *true* by p_i to indicate that it wants to access the resource. In that way, when a process releases the resource, it knows which processes want to use it.
- An array of Boolean SWMR registers *PRIORITY*$[1..n]$. The register *PRIORITY*$[i]$ is used by p_i to store the priority of its current request (if any).

Finally, the waiting room is implemented as follows:

- An array of semaphores *SLEEP_CHAIR*$[1..n]$ initialized to 0.

  When a process p_i has to wait before using the resource, it "goes to sleep" on its personal sleeping chair until it is woken up. "Going to sleep" is implemented by invoking *SLEEP_CHAIR*$[i]$.down() (let us remember that this semaphore is initialized to 0), and another process wakes p_i up by invoking *SLEEP_CHAIR*$[i]$.up().

It is important to see the role of the pair (*WAITING*$[i]$, *SLEEP_CHAIR*$[i]$) for each process p_i. The value of the Boolean *WAITING*$[i]$ is written on the blackboard and consequently allows a process p_j that reads the blackboard to know if p_i is waiting for the resource and, according to the priority, to wake it up.

The operations acquire() **and** release() The algorithms implementing acquire() and release() for a process p_i are described in Fig. 3.9.

```
operation acquire($i, prio_i$) is
(1)  $MUTEX$.down();
(2)  if ($BUSY$) then $WAITING[i] \leftarrow true$; $PRIORITY[i] \leftarrow prio_i$;
(3)                   $MUTEX$.up();
(4)                   $SLEEP\_CHAIR[i]$.down();
(5)              else $BUSY \leftarrow true$;
(6)                   $MUTEX$.up()
(7)  end if;
(8)  return()
end operation.

operation release($i$) is
(9)  $MUTEX$.down();
(10) if ($\exists j : WAITING[j]$) then let $p_k$ = waiting process with the highest priority;
(11)                                    $WAITING[k] \leftarrow false$;
(12)                                    $SLEEP\_CHAIR[k]$.up();
(13)                               else $BUSY \leftarrow false$
(14) end if;
(15) $MUTEX$.up();
(16) return()
end operation.
```

Fig. 3.9 Resource allocation with priority (code for process p_i)

When a process p_i invokes acquire(i), it has first to obtain exclusive access to the blackboard (invocation $MUTEX$.down(), line 1). After it has obtained the mutual exclusion, there are two cases:

- If the resource is free (test at line 2), before proceeding to use the resource, p_i sets $BUSY$ to $true$ to indicate to the other processes that the resource is no longer available (line 5) and releases the mutual exclusion on the blackboard (line 6).
- If the resource is not available, p_i goes to sleep on its sleeping chair (line 4). But before going to sleep, p_i must write on the blackboard the priority associated with its request and the fact that it is waiting for the resource (line 2). Before going to sleep, p_i has also to release the mutual exclusion on the blackboard so that other processes can access it (line 3).

To release the resource, a process p_i invokes release(i), which does the following. First p_i requires mutual exclusion on the blackboard in order to read consistent values (line 9). This mutual exclusion will be released at the end of the operation (line 15). After it has obtained exclusive access to the blackboard, p_i checks if there are processes waiting for the resource (predicate $\exists j \neq i : WAITING[j]$, line 10). If no process is waiting, p_i resets $BUSY$ to $false$ (line 13) to indicate to the next process that invokes acquire() that the resource is available. Otherwise, there are waiting processes: $\{j \mid WAITING[j]\} \neq \emptyset$. In that case, after it has determined the process p_k whose request has the strongest priority among all waiting processes (line 10), p_i resets $WAITING[k]$ to $false$ (line 11) and wakes up p_k (line 12) before exiting the blackboard room (line 15).

Remark Let us observe that, due to asynchrony, it is possible that a process p_i wakes up a waiting process p_k (by executing *SLEEP_CHAIR*[k].up(), line 12), before p_k has executed *SLEEP_CHAIR*[k].down() (this can occur if p_k spends a long time to go from line 3 to line 4). The reader may check that this does not cause problems: actually, the slowness of p_k between line 3 and line 4 has the same effect as if p_k was waiting for *SLEEP_CHAIR*[k] to become positive.

On the way priorities are defined The previous resource allocation algorithm is very general. The way priorities are defined does not depend on it. Priorities can be statically associated with processes, or can be associated with each invocation of the operation acquire() independently one from another.

A token metaphor The algorithms in Fig. 3.9 can be seen as a token management algorithm. There is initially a single token deposited on a table (this is expressed by the predicate $BUSY = false$).

When there is no competition a process takes the token which is on the table (statement $BUSY \leftarrow true$, line 5), and when it has finished using the resource, it deposits it on the table (statement $BUSY \leftarrow false$, line 13).

When there is competition, the management of the token is different. The process p_i that owns the token gives it directly to a waiting process p_k (statement *SLEEP_CHAIR*[i].up(), line 12) and p_k obtains it when it terminates executing *SLEEP_CHAIR*[i].down() (line 4). In that case, due to priorities, the transmission of the token is "direct" from p_i to p_k. The Boolean register *BUSY* is not set to *true* by p_i, and after being reset to *false* by p_k, it remains equal to *false* until p_k gives the token to another process or deposits it on the table at line 13 if no process wants to access the resource.

3.2.4 Using Semaphores to Solve the Readers-Writers Problem

The readers-writers problem Let us consider a file that can be accessed by a single process by the operations read_file() and write_file(). The readers-writers problem consists in designing a concurrent object that allows several processes to access this file in a consistent way. Consistent means here that any number of processes can simultaneously read the file, but at most one process at a time can write the file, and writing the file and reading the file are mutually exclusive.

To that end, an approach similar to the one used in Fig. 3.1 to go from a sequential stack to a concurrent stack can be used. As reading a file does not modify it, using a lock would be too constraining (as it would allow at most one read at a time). In order to allow several read operations to execute simultaneously, let us instead design a concurrent object defined by four operations denoted begin_read(), end_read(), begin_write(), and end_write(), and let us use them to bracket the operation read_file() and write_file(), respectively. More precisely, the high-level operations used by the processes, denoted conc_read_file() and conc_write_file(), are defined as described in Fig. 3.10.

```
operation conc_read_file() is
    begin_read(); read_file(); end_read(); return()
end operation.

operation conc_write_file($v$) is
    begin_write(); write_file($v$); end_write(); return()
end operation.
```

Fig. 3.10 From a sequential file to a concurrent file

Remark It is easy to see that the readers-writers problem is a generalization of the mutual exclusion problem. If there was only the write operation, the problem would be the same as mutual exclusion. The generalization consists in having two classes of operations (read and write), with specific execution constraints in each class (mutual exclusion among the write operations and no constraint among the read operations) and constraints among the two classes (which are mutually exclusive).

Readers-writers with weak priority to the readers A semaphore *GLOBAL_ MUTEX* initialized to 1 can be used to ensure mutual exclusion on the write operations. Moreover the same semaphore can be used to ensure mutual exclusion between each write operation and the concurrent read operations. Such a solution is presented in Fig. 3.11.

The code of begin_write() and end_write() (lines 11–14) follows from the previous observation. As far as the read operations are concerned, a shared register, denoted *NBR* and initialized to 0, is introduced to count the current number of concurrent readers. This register is increased each time begin_read() is invoked (line 2) and decreased each time end_read() is invoked (line 7). Moreover, to keep the value of *NBR* consistent, its read and write accesses are done in mutual exclusion. This is implemented by the semaphore *NBR_MUTEX* initialized to 1 (lines 1, 4, 6 and 9).

The mutual exclusion semaphore *GLOBAL_MUTEX* is used as follows by the readers. A "first" reader (i.e., a reader that finds $NBR = 1$) invokes *GLOBAL_MUTEX*.down() (line 3) to obtain an exclusive access to the file, not for it, but for the class of all readers. This mutually exclusive access will benefit for free to all the readers that will arrive while $NBR > 1$. Similarly, a "last" reader (i.e., a reader that finds $NBR = 0$) invokes *GLOBAL_MUTEX*.up() (line 8) to release the mutual exclusion on the file that was granted to the class of readers.

If at least one process is reading the file, we have $NBR \geq 1$, which gives priority to the class of readers. As we have seen, this is because a reader obtains mutual exclusion nor for itself but for the class of readers (an invocation of begin_read() does not need to invoke *GLOBAL_MUTEX* when $NBR \geq 1$). It follows that writers can be blocked forever if permanently readers invoke begin_read() while $NBR \geq 1$. The read operation consequently has priority over the write operation, but this is a *weak* priority as shown below.

Readers-writers with strong priority to the readers When considering the previous algorithm, let us consider the case where at least two processes p_{w1}

```
operation begin_read() is
(1)   $NBR\_MUTEX$.down();
(2)      $NBR \leftarrow NBR + 1$;
(3)      if $(NBR = 1)$ then $GLOBAL\_MUTEX$.down() end if;
(4)   $NBR\_MUTEX$.up();
(5)   return()
end operation.

operation end_read() is
(6)   $NBR\_MUTEX$.down();
(7)      $NBR \leftarrow NBR - 1$;
(8)      if $(NBR = 0)$ then $GLOBAL\_MUTEX$.up() end if;
(9)   $NBR\_MUTEX$.up();
(10)  return()
end operation.

operation begin_write() is
(11)  $GLOBAL\_MUTEX$.down();
(12)  return()
end operation.

operation end_write() is
(13)  $GLOBAL\_MUTEX$.up();
(14)  return()
end operation.
```

Fig. 3.11 Readers-writers with weak priority to the readers

and p_{w2} have invoked begin_write() and one of them (say p_{w1}) has obtained the mutual exclusion to write the file. *GLOBAL_MUTEX* is then equal to 0 and the other writers are blocked at line 11. Let us assume that a process p_r invokes begin_read(). The counter *NBR* is increased from 0 to 1, and consequently p_r invokes *GLOBAL_MUTEX*.down() and becomes blocked. Hence, p_{w2} and p_r are blocked on the semaphore *GLOBAL_MUTEX* with p_{w2} blocked before p_r. Hence, when later p_{w1} executes end_write(), it invokes *GLOBAL_MUTEX*.up(), which unblocks the first process blocked on that semaphore, i.e., p_{w2}.

Strong priority to the read operation is weak priority (a reader obtains the priority for the class of readers) plus the following property: when a writer terminates and readers are waiting, the readers have to immediately obtain the mutual exclusion. As already noticed, the readers-writers object described in Fig. 3.11 satisfies weak priority for the readers but not strong priority.

There is a simple way to enrich the previous implementation to obtain an object implementation that satisfies strong priority for the readers. It consists in ensuring that, when several processes invoke begin_write(), at most one of them is allowed to access the semaphore *GLOBAL_MUTEX* (in the previous example, p_{w2} is allowed to invoke *GLOBAL_MUTEX*.down() while p_{w1} had not yet invoked *GLOBAL_MUTEX*.up()). To that end a new semaphore used only by the writers is introduced. As its aim is to ensure mutual exclusion among concurrent writers, this semaphore, which is denoted *WRITER_MUTEX*, is initialized to 1.

```
operation begin_write() is
(11.0)   $WRITER\_MUTEX$.down();
(11)     $GLOBAL\_MUTEX$.down();
(12)     return()
end operation.

operation end_write() is
(13)     $GLOBAL\_MUTEX$.up();
(13.0)   $WRITER\_MUTEX$.up();
(14)     return()
end operation.
```

Fig. 3.12 Readers-writers with strong priority to the readers

The implementation of the corresponding object is described in Fig. 3.12. As the algorithms for the operations begin_read() and end_read() are the same as in Fig. 3.11, they are not repeated here. The line number of the new lines is postfixed by "0". It is easy to see that at most one writer at a time can access the semaphore *GLOBAL_MUTEX*.

Readers-writers with priority to the writers Similarly to the weak priority to the readers, weak priority to the writers means that, when a writer obtains the right to write the file, it obtains it not for itself but for all the writers that arrive while there is a write that is allowed to access the file.

The corresponding readers-writers concurrent object is described in Fig. 3.13. Its structure is similar to the one of Fig. 3.11. Moreover, the same lines in both figures have the same number.

The writers now have to count the number of concurrent write invocations. A counter denoted *NBW* is used to that end. The accesses to that register (which is shared only by the writers) are protected by the semaphore *NBW_MUTEX* initialized to 1.

The corresponding lines are numbered NW.1–NW.4 in begin_write() and NW.5–NW.8 in end_write(). Moreover a new semaphore, denoted *PRIO_W_MUTEX* and initialized to 1, is now used to give priority to the writers. This semaphore is used in a way similar to the way the semaphore *GLOBAL_MUTEX* is used in Fig. 3.11 to give weak priority to the readers.

While *GLOBAL_MUTEX* ensures mutual exclusion between any pair of write operations and between any write operation and read operations, *PRIO_W_MUTEX* allows the "first" writer that is allowed to access the file to obtain priority not for itself but for the class of writers (lines NW.3 and NW.7). To attain this priority goal, the operation begin_read() now has to be bracketed by *PRIO_W_MUTEX*.down() at its beginning (line NR.1) and *PRIO_W_MUTEX*.up() at its end (line NR.2). Hence, each read now competes with all concurrent writes considered as a single operation, which gives weak priority to the readers.

```
operation begin_read() is
(NR.1)  $PRIO\_W\_MUTEX$.down();
(1)     $NBR\_MUTEX$.down();
(2)        $NBR \leftarrow NBR + 1$;
(3)        if $(NBR = 1)$ then $GLOBAL\_MUTEX$.down() end if;
(4)     $NBR\_MUTEX$.up();
(NR.2)  $PRIO\_W\_MUTEX$.up();
(5)     return()
end operation.

operation end_read() is
(6)     $NBR\_MUTEX$.down();
(7)        $NBR \leftarrow NBR - 1$;
(8)        if $(NBR = 0)$ then $GLOBAL\_MUTEX$.up() end if;
(9)     $NBR\_MUTEX$.up();
(10)    return()
end operation.

operation begin_write() is
(NW.1)  $NBW\_MUTEX$.down();
(NW.2)     $NBW \leftarrow NBW + 1$;
(NW.3)     if $(NBW = 1)$ then $PRIO\_W\_MUTEX$.down() end if;
(NW.4)  $NBW\_MUTEX$.up();
(11)    $GLOBAL\_MUTEX$.down();
(12)    return()
end operation.

operation end_write() is
(13)    $GLOBAL\_MUTEX$.up();
(NW.5)  $NBW\_MUTEX$.down();
(NW.6)     $NBW \leftarrow NBW - 1$;
(NW.7)     if $(NBW = 0)$ then $PRIO\_W\_MUTEX$.up() end if;
(NW.8)  $NBW\_MUTEX$.up();
(14)    return()
end operation.
```

Fig. 3.13 Readers-writers with priority to the writers

3.2.5 Using a Buffer to Reduce Delays for Readers and Writers

When considering the readers-writers problem, the mutual exclusion used in the previous solutions between base read and write operations can entail that readers and writers experience long delays when there is heavy contention for the shared file. This section presents a solution to the readers-writers problem that reduces waiting delays. Interestingly, this solution relies on the producer-consumer problem.

Read/write lock A read/write lock is a synchronization object that (a) provides the processes with the four operations begin_read(), end_read(), begin_write(), and end_write() and (b) ensures one of the specifications associated with the readers-writers problem (no priority, weak/strong priority to the readers or the writers, etc.).

Semaphore-based implementations of such objects have been presented previously in Sect. 3.2.4.

The case of one writer and several readers The principle is fairly simple. There is a buffer with two entries $BUF[0..1]$ that is used alternately by the writer and a reader reads the last entry that was written. Hence, it is possible for a reader to read a new value of the file while the writer is writing a value that will become the new "last" value.

While, for each $x \in \{0, 1\}$, several reads of $BUF[x]$ can proceed concurrently, writing into $BUF[x]$ and reading from $BUF[x]$ have to be mutually exclusive. To that end, a read/write lock $RW\_LOCK[x]$ is associated with each $BUF[x]$.

The write of a new value v into $BUF[x]$ is denoted $BUF[x]$.write(v) and a read from $BUF[x]$ is denoted $r \leftarrow BUF[x]$.read() (where r is a local variable of the invoking process). Initially, both $BUF[0]$ and $BUF[1]$ contain the initial value of the file.

In addition to $BUF[0..1]$, the processes share an atomic register denoted $LAST$ that contains the index of the last buffer that was written by the writer. It is initialized to 0.

The algorithms implementing the conc_read_file() and conc_write_file() operations of the corresponding readers-writer object are described in Fig. 3.14.

As previously indicated, a read returns the last value that was written, and a write deposits the new value in the other entry of the buffer which becomes the "last" one when the write terminates. It is important to see that, differently from the producer-consumer problem, not all the values written are necessarily read.

An execution is described in Fig. 3.15, which considers the writer and a single reader. Its aim is to show both the efficiency gain and the mutual exclusion requirement on each buffer entry. The efficiency gain appears with the first read operation: this operation reads from $BUF[1]$ while there is a concurrent write into

```
operation conc_read_file() is
(1)      $my\_index \leftarrow LAST$;
(2)      $RW\_LOCK[my\_index]$.begin_read();
(3)         $r \leftarrow BUF[my\_index]$.read();
(4)      $RW\_LOCK[my\_index]$.end_read();
(5)      return($r$)
end operation.

operation conc_write_file($v$) is
(6)      $new\_last \leftarrow (LAST + 1) \mod 2$;
(7)      $RW\_LOCK[new\_last]$.begin_write();
(8)         $BUF[new\_last]$.write($v$);
(9)      $RW\_LOCK[new\_last]$.end_write();
(10)     $LAST \leftarrow new\_last$;
(11)     return()
end operation.
```

Fig. 3.14 One writer and several readers from producer-consumer

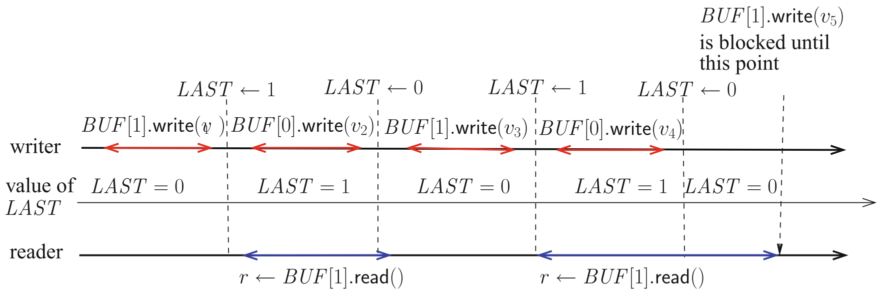


Fig. 3.15 Efficiency gain and mutual exclusion requirement

BUF[0]. Similarly, the last read (which is of *BUF*[1] because *LAST* = 1 when the corresponding conc_read_file() operation starts) is concurrent with the write into *BUF*[0]. Hence, the next write operation (namely conc_write_file(v_5)) will be on *BUF*[1]. If conc_write_file(v_5) is invoked while *BUF*[1].read() has not terminated, the write must be constrained to wait until this read terminates. Hence, the mutual exclusion requirement on the reads and writes on each entry of the buffer.

Starvation-freedom There are two algorithm instances that ensure the mutual exclusion property: one between *BUF*[0].read() and *BUF*[0].write() and the other one between *BUF*[1].read() and *BUF*[1].write().

Let us assume that these algorithms guarantee the starvation-freedom property for the read operations (i.e., each invocation of *BUF*[*x*].read() terminates). Hence, there is no specific liveness property attached to the base *BUF*[0].write() operations. The following theorem captures the liveness properties of the conc_read_file() and conc_write_file() operations which are guaranteed by the algorithms described in Fig. 3.14.

Theorem 12 *If the underlying read/write lock objects ensure starvation-freedom for the read operations, then the implementation of the operations* conc_read_file() *and* conc_write_file() *given in Fig. 3.14 ensure starvation-freedom for both operations.*

Proof Starvation-freedom of the invocations of the operation conc_read_file() follows trivially from the starvation-freedom of the base *BUF*[*x*].read() operation.

To show that each invocation of the operation conc_write_file() terminates, we show that any invocation of the operation *BUF*[*x*].write() does terminate. Let us consider an invocation of *BUF*[*x*].write(). Hence, *LAST* = $1 - x$ (lines 6 and 10). This means that the read invocations that start after the invocation *BUF*[*x*].write() are on *BUF*[$1-x$]. Consequently, these read invocations cannot prevent *BUF*[*x*].write() from terminating.

Let us now consider the invocations *BUF*[*x*].read() which are concurrent with *BUF*[*x*].write(). There is a finite number of such invocations. As the underlying mutual exclusion algorithm guarantees starvation for these read invocations, there a finite time after which they all have terminated. If the invocation *BUF*[*x*].write()

```
operation conc_read_file() is
(1)      $my\_index \leftarrow LAST$;
(2)      $RW\_LOCK[my\_index]$.begin_read();
(3)          $r \leftarrow BUF[my\_index]$.read();
(4)      $RW\_LOCK[my\_index]$.end_read();
(5)      return($r$)
end operation.

operation conc_write_file($i, v$) is
(7)      $RW\_LOCK[my\_last]$.begin_write();
(8)          $BUF[my\_last]$.write($v$);
(9)      $RW\_LOCK[my\_last]$.end_write();
(10)     $LAST \leftarrow my\_last$;
(NW.1)   $my\_last \leftarrow my\_last + 1$;
(NW.2)   if ($my\_last = 2i$) then $my\_last \leftarrow 2i - 2$ end if;
(11)     return()
end operation.
```

Fig. 3.16 Several writers and several readers from producer-consumer

has not yet been executed, it is the only operation on $BUF[x]$ and is consequently executed, which concludes the proof of the theorem. □

The case of several writers and several readers The previous single writer/multi-reader algorithm can be easily generalized to obtain a multi-writer/multi-reader algorithm.

Let us assume that there are m writers denoted $q_1, \ldots, q_m$. The array of registers now has $2m$ entries: $BUF[0..(2m-1)]$, and the registers $BUF[2i-2]$ and $BUF[2i-1]$ are associated with the writer number q_i. As previously, a writer q_i writes alternately $BUF[2i-2]$ and $BUF[2i-1]$ and updates *LAST* after it has written a base register. The corresponding write algorithm is described in Fig. 3.16. The local index *my_last* of q_i is initialized to $2i-2$. Basically line 6 of Fig. 3.14 is replaced by the new lines NW.1–NW.2. Moreover, the local variable *new_last* is now renamed *my_last* and the algorithm for the operation conc_read_file() is the same as before.

3.3 A Construct for Imperative Languages: the Monitor

Semaphores are synchronization objects that allow for the construction of application-oriented concurrent objects. Unfortunately, they are low-level counting objects.

The concept of a *monitor* allows for the definition of concurrent objects at a "programming language" abstraction level. Several variants of the monitor concept have been proposed. This concept was developed by P. Brinch Hansen and C.A.R. Hoare from an initial idea of E.W. Dijkstra. To introduce it, this section adopts Hoare's presentation (1974).

3.3.1 The Concept of a Monitor

A monitor is an object A monitor is a concurrent object. Hence, it offers operations to the processes, and only these operations can access its internal representation.

The mutual exclusion guarantee While its environment is made up of parallel processes, a monitor object guarantees mutual exclusion on its internal representation: at most one operation invocation at a time can be active inside the monitor. This means that, when a process is executing a monitor operation, it has a consistent view of the monitor internal representation, as no other process can concurrently be active inside the monitor.

As an example let us consider a resource that has to be accessed in mutual exclusion. A simple monitor-based solution consists in defining a monitor which contains the resource and defining a single operation use_resource() that the monitor offers to processes.

The queues (conditional variables) In order to solve issues related to internal synchronization (e.g., when a process has to wait for a given signal from another process), the monitor concept provides the programmer with specific objects called *conditions*. A condition C is an object that can be used only inside a monitor. It offers the following operations to the processes:

- Operation C.wait().
  When a process p invokes C.wait() it stops executing and from an operational point of view it waits in the queue C. As the invoking process is no longer active, the mutual exclusion on the monitor is released.
- Operation C.signal().
  When a process p invokes C.signal() there are two cases according to the value of C:
  - If no process is blocked in the queue C, the operation C.signal() has no effect.
  - If at least one process is blocked in the queue C, the operation C.signal() reactivates the first process blocked in C. Hence, there is one fewer process blocked in C but two processes are now active inside the monitor. In order to guarantee that a single process at a time can access the internal representation of the monitor the following rule is applied:
    - * The process which was reactivated becomes active inside the monitor and executes the statements which follows its invocation C.wait().
    - * The process that has executed C.signal() becomes passive but has priority to re-enter the monitor. When allowed to re-enter the monitor, it will execute the statements which follow its invocation C.signal().
- Operations C.empty().
  This operation returns a Boolean value indicating if the queue C is empty or not.

3.3.2 A Rendezvous Monitor Object

In order to illustrate the previous definition, let us consider the implementation of a rendezvous object.

Definition As indicated in Chap. 1, a rendezvous object is associated with *n* control points, one in each process participating in the rendezvous. It offers them a single operation denoted rendezvous() (or barrier()). (The control point of a process is the location in its control flow where it invokes rendezvous().)

The semantics of the rendezvous object is the following: any process involved in the rendezvous can pass its control point (terminate its rendezvous() operation) only when all other processes have attained their control point (invoked their rendezvous() operation). Operationally speaking, an invocation of the rendezvous() operation blocks the invoking process until all the processes involved in the rendezvous have invoked the rendezvous() operation.

As already noticed, a rendezvous object cannot be specified with a sequential specification.

An atomic register-based implementation Let us consider a register denoted *COUNTER* that can be atomically accessed by three primitives, read, write, and fetch&add() (which atomically adds 1 to the corresponding register). The domain of *COUNTER* is the set $\{0, 1, \ldots, n\}$.

An implementation of a rendezvous object based on such a counter register, initialized to 0, and a flag, denoted *FLAG*, is described in Fig. 3.17. *FLAG* is an atomic binary register. Its initial value is any value (e.g., 0 or 1). Moreover, let $\overline{FLAG}$ denote the "other" value (i.e., the value which is not currently stored in *FLAG*).

A process that invokes rendezvous() first reads the value of *FLAG* and stores it in a local variable *flag* (line 1) and increases *COUNTER* (line 2). The register *COUNTER* counts the number of processes that have attained their control point (i.e., invoked rendezvous()). Hence, if $COUNTER = m$, each of the m processes involved in the rendezvous has attained its control point. In that case, p_i resets *COUNTER* to 0 and *FLAG* is switched to $\overline{FLAG}$ to signal that each of the m processes has attained its control point (line 3). Differently, if $COUNTER \neq m$, p_i enters a waiting loop controlled by the predicate $flag \neq FLAG$ (line 4). As just seen, this predicate becomes true when *COUNTER* becomes equal to m.

```
operation rendezvous() is
(1)  $flag \leftarrow FLAG$;
(2)  $COUNTER$.fetch&add();
(3)  if ($COUNTER = m$) then $COUNTER \leftarrow 0$; $FLAG \leftarrow \overline{FLAG}$
(4)                     else wait($flag \neq FLAG$)
(5)  end if;
(6)  return()
end operation.
```

Fig. 3.17 A register-based rendezvous object

Let us observe that a rendezvous object implemented as in Fig. 3.17 is not restricted to be used only once. It can be used repeatedly to synchronize a given set of processes as many times as needed. We say that the rendezvous object is not restricted to be a *one-shot* object.

A monitor-based rendezvous The internal representation of the rendezvous object (for m participating processes) consists of a register denoted *COUNTER* (initialized to 0) and a condition denoted *QUEUE*.

The algorithm implementing the operation rendezvous() is described in Fig. 3.18. Let us remember that, when a process is active inside the monitor, it is the only active process inside the monitor. Hence, the algorithm implementing the operation rendezvous() can be designed as a sequential algorithm which momentarily stops when it executes *QUEUE*.wait() and restarts when it is reactivated by another process that has executed *QUEUE*.signal(). As we are about to see, as an invocation of *QUEUE*.signal() reactivates at most one process, the only tricky part is the management of the invocations of *QUEUE*.signal() so that all blocked processes are eventually reactivated.

When one of the m participating processes invokes rendezvous(), it first increases *COUNTER* (line 1) and then checks the value of *COUNTER* (line 2). If $COUNTER < m$, it is blocked and waits in the condition *QUEUE* (line 2). It follows that the $(m-1)$ first processes which invoke rendezvous() are blocked and wait in that condition. The mth process which invokes rendezvous() increases *COUNTER* to m and consequently resets *COUNTER* to 0 (line 3) and reactivates the first process that is blocked in the condition *QUEUE*. When reactivated, this process executes *QUEUE*.signal() (line 5), which reactivates another process, etc., until all m processes are reactivated and terminate their invocations of rendezvous().

Let us notice that, after their first rendezvous has terminated, the m processes can use again the very same object for a second rendezvous (if needed). As an example, this object can be used to re-synchronize the processes at the beginning of each iteration of a parallel loop.

```
monitor $RDV$ is
   $COUNTER \in \{0, 1, \ldots, m\}$ init 0;
   $QUEUE$ condition.

  operation rendezvous() is
  (1) $COUNTER \leftarrow COUNTER + 1$;
  (2) if $(COUNTER < m)$ then $QUEUE$.wait();
  (3)                         else $COUNTER \leftarrow 0$
  (4) end if;
  (5) $QUEUE$.signal();
  (6) return()
  end operation.
end monitor.
```

Fig. 3.18 A monitor-based rendezvous object

3.3.3 Monitors and Predicates

A simple producer-consumer monitor A monitor encapsulating a buffer of size k accessed by a single producer and a single consumer is described in Fig. 3.19. The internal representation consists of an array $BUF[0..(k-1)]$ and the two variables *in* and *out* (as used previously) plus a shared register *NB_FULL* counting the number of values produced and not yet consumed and two conditions (queues): *C_PROD*, which is the waiting room used by the producer when the buffer is full, and *C_CONS*, which is the waiting room used by the consumer when the buffer is empty.

When the producer invokes produce(v) it goes to wait in the condition (queue) *C_PROD* if the buffer is full (line 1). When it is reactivated, or immediately if the buffer is not full, it writes its value in the next buffer entry (as defined by *in*, line 2) and increases *NB_FULL* (line 3). Finally, in case the consumer is waiting for a production, it invokes *C_CONS*.signal() (line 4) to reactivate it (let us remember that this invocation has no effect if the consumer is not blocked in *C_CONS*). The algorithm implementing consume() is similar.

A more general and efficient monitor Inspired from the implementation given in Fig. 3.7, a more efficient monitor could be designed that would allow several producers and several consumers to concurrently access distinct entries of the array $BUF[1..(k-1)]$. Such a monitor would provide operations begin_produce() and end_produce() that would allow a producer to bracket the write of its value, and

```
monitor $SIMPLE\_SPSC$ is
    $BUF[0..(k-1)]$ of item values;
    $in, out \in \{0, 1, \ldots, k-1\}$ init 0;
    $NB\_FULL \in \{0, 1, \ldots, k\}$ init 0;
    $C\_PROD, C\_CONS$: condition.

  operation produce($v$) is
   (1)  if $(NB\_FULL = k)$ then $C\_PROD$.wait() end if;
   (2)  $BUF[in]$.write($v$); $in \leftarrow (in+1) \mod k$:
   (3)  $NB\_FULL \leftarrow NB\_FULL + 1$;
   (4)  $C\_CONS$.signal();
   (5)  return()
  end operation.

  operation consume() is
   (6)  if $(NB\_FULL = 0)$ then $C\_CONS$.wait() end if;
   (7)  $r \leftarrow BUF[out]$.read(); $out \leftarrow (out+1) \mod k$:
   (8)  $NB\_FULL \leftarrow NB\_FULL - 1$;
   (9)  $C\_PROD$.signal();
   (10) return($r$)
  end operation.
end monitor.
```

Fig. 3.19 A simple single producer/single consumer monitor

operations begin_consume() and end_consume() that would allow a consumer to bracket the read of a value. Due to the monitor semantics, these four operations will be executed in mutual exclusion.

Why signal() **entails the blocking of the invoking process** In the following we consider only the case where the producer is blocked and the consumer reactivates it. The same applies to the case where the consumer is blocked and the producer reactivates it.

The producer has to be blocked when the buffer is full (line 1 in Fig. 3.19). Said differently, to keep the buffer correct, the relation $I \equiv (0 \leq NB\_FULL \leq k)$ has to be kept invariant (let us remember that synchronization is to preserve invariants). This has two consequences:

- First, as it increases NB_FULL, the progress of the producer is constrained by the condition $P \equiv (NB\_FULL < k)$, which means that the producer has to be blocked when it executes C_PROD.wait() if its progress violates I.
- Second, when P becomes true (and in the producer-consumer problem this occurs each time the consumer has consumed an item), the fact that control of the monitor is immediately given to the process that it has reactivated guarantees that P is satisfied when the reactivated process continues its execution.

This is sometimes called a *transfer of a predicate* (Fig. 3.20). More generally, if a process p that has invoked C.signal() were to continue executing inside the monitor, it would be possible for the predicate P to have been falsified when the reactivated process accesses the monitor. Forcing C.signal() to pass the control of the monitor to the reactivated process (if any) is a simple way to preserve the invariant associated with a monitor.

Signal-and-wait versus signal-and-continue Let $C[P]$ be the condition (queue) associated with a predicate P. This means that if, when evaluated by a process, the predicate P on the internal representation of the monitor is false, this process is directed to wait in the queue $C[P]$.

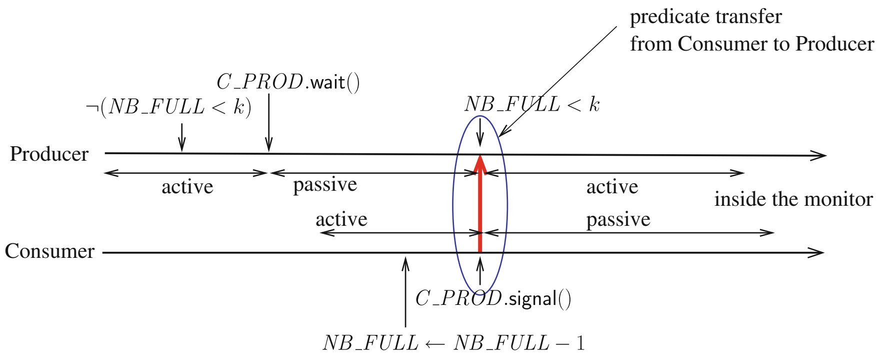


Fig. 3.20 Predicate transfer inside a monitor

The previous semantics is called *signal-and-wait*. Another monitor variant uses the semantics called *signal-and-continue*; namely, the process that invokes $C[P]$.signal() continues its execution inside the monitor and the process that has been reactivated (if any) then has priority to enter the monitor and execute the statements that follow its invocation of $C[P]$.wait().

The main difference between a signal-and-wait monitor and a signal-and-continue monitor is the following. The statement

if $(\neg P)$ **then** $C[P]$.wait() **end if**

used in the signal-and-wait monitor has to be replaced by the statement

while $(\neg P)$ **do** $C[P]$.wait() **end while**

in a signal-and-continue monitor. It is easy to see that, in both cases, the reactivated process is allowed to progress only if the predicate P is satisfied, thereby ensuring correct predicate transfer.

3.3.4 Implementing a Monitor from Semaphores

The base objects (semaphores and integers) used to implement the control part of a signal-and-wait monitor are described below. This monitor internal structure is depicted in Fig. 3.21 (let us remark that the control part of the internal structure is similar to the structure depicted in Fig. 3.8).

- A semaphore *MUTEX*, initialized to 1, is used to ensure mutual exclusion on the monitor (at most one process at a time can access the monitor internal representation).

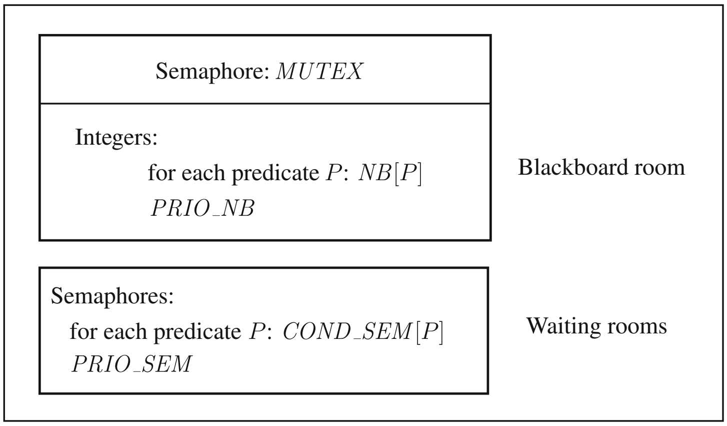


Fig. 3.21 Base objects to implement a monitor

- A semaphore denoted *COND_SEM*[*P*] and an integer *NB*[*P*] are associated with each pair made up of a waiting predicate *P* and its condition *C*[*P*].

  The semaphore *COND_SEM*[*P*], which is initialized to 0, is used as a waiting room for the processes that have invoked *C*[*P*].wait(). The register *NB*[*P*], initialized to 0, counts the number of processes currently waiting in this room.
- A semaphore *PRIO_SEM*, initialized to 0, is used as a waiting room for the processes that have invoked *C*[*P*].signal(), whatever the condition *C*[*P*]. As already indicated, the processes blocked in *PRIO_SEM* have priority to re-enter the monitor. The integer *PRIO_NB*, initialized to 0, counts the number of processes blocked on the semaphore *PRIO_SEM*.

A monitor implementation, based on the previous integers and semaphores, is described in Fig. 3.22. It is made up of four parts.

When a process *p* invokes an operation of the monitor, it has to require mutual exclusion on its internal representation and consequently invokes *MUTEX*.down() (line 1). When it terminates a monitor operation, there are two cases according to whether there are or are not processes that are blocked because they have executed a statement *C*[*P*].signal() (whatever *P*). If there are such processes, we have $PRIO\_NB > 0$ (see line 11). Process *p* then reactivates the first of them by invoking

```
when a process invokes a monitor operation do
(1)   $MUTEX$.down()
end do.

when a process terminates a monitor operation do
(2)   if $(PRIO\_NB > 0)$ then $PRIO\_SEM$.up()
(3)                         else $MUTEX$.up()
(4)   end if
end do.

when a process invokes $C[P]$.wait() do
(5)   $NB[P] \leftarrow NB[P] + 1$;
(6)   if $(NB[P] > 0)$ then $PRIO\_SEM$.up()
(7)                     else $MUTEX$.up()
(8)   end if;
(9)   $COND\_SEM[P]$.down();
(10)  $NB[P] \leftarrow NB[P] - 1$
end do.

when a process invokes $C[P]$.signal() do
(11)  if $(NB[P] > 0)$ then $PRIO\_NB \leftarrow PRIO\_NB + 1$;
(12)                         $COND\_SEM[P]$.up();
(13)                         $PRIO\_SEM$.down();
(14)                         $PRIO\_NB \leftarrow PRIO\_NB - 1$
(15)  end if
end do.
```

Fig. 3.22 Semaphore-based implementation of a monitor

PRIO_SEM.up() (line 2). Hence, the control inside the monitor passes directly from p to the reactivated process. If there are no such processes, we have $PRIO\_NB = 0$. In that case p releases the mutual exclusion on the monitor by releasing the semaphore *MUTEX* (line 3).

The code executed by a process p that invokes $C[P]$.wait() is described at lines 5–10. Process p then has to be blocked on the semaphore $COND\_SEM[P]$, which is the waiting room associated with the condition $C[P]$ (line 9). Hence, p increases $NB[P]$ before being blocked (line 5) and will decrease it when reactivated (line 10). Moreover, as p is going to wait, it has to release the mutual exclusion on the monitor internal representation. If there is a process q which is blocked due to an invocation of wait(), it has priority to re-enter the monitor. Consequently, p directly passes control of the monitor to q (line 6). Differently, if no process is blocked on *PRIO_SEM*, p releases the mutual exclusion on the monitor entrance (line 7) to allow a process that invokes a monitor operation to enter it.

The code executed by a process p that invokes $C[P]$.signal() is described at lines 11–15. If no process is blocked in the condition $C[P]$ we have $NB[P] = 0$ and there is nothing to do. If $NB[P] > 0$, the first process blocked in $C[P]$ has to be reactivated and p has to become a priority process to obtain the control of the monitor again. To that end, p increases *PRIO_NB* (line 11) and reactivates the first process blocked in $C[P]$ (line 12). Then, it exits the monitor and goes to wait in the priority waiting room *PRIO_SEM* (line 13). When later reactivated, it will decrease *PRIO_NB* in order to indicate that one fewer process is blocked in the priority semaphore *PRIO_SEM* (line 14).

3.3.5 Monitors for the Readers-Writers Problem

This section considers several monitors suited to the readers-writers problem. They differ in the type of priority they give to readers or writers. This family of monitors gives a good illustration of the programming comfort provided by the monitor construct (the fact that a monitor allows a programmer to use directly the power of a programming language makes her/his life easier).

These monitors are methodologically designed. They systematically use the following registers (which are all initialized to 0 and remain always non-negative):

- *NB_WR* and *NB_AR* denote the number of readers which are currently waiting and the number of readers which are currently allowed to read the file, respectively.
- *NB_WW* and *NB_AW* denote the number of writers which are currently waiting and the number of writers which are currently allowed to write the file, respectively.

In some monitors that follow, *NB_WR* and *NB_AR* could be replaced by a single register *NB_R* (numbers of readers) whose value would be $NB\_WR + NB\_AR$ and, as its value is 0 or 1, *NB_AW* could be replaced by a Boolean register. This is not done to insist on the systematic design dimension of these monitors.

A readers-writers monitor encapsulates the operations begin_read() and end_read() which bracket any invocation of read_file() and the operations begin_write() and end_write() which bracket any invocation of write_file() as described in Fig. 3.10 (Sect. 3.2.4).

Readers-writers invariant The mutual exclusion among the writers means that (to be correct) a monitor has to be always such that $0 \leq NB\_AW \leq 1$. The mutual exclusion between the readers and the writers means that it has also to be always such that $(NB\_AR = 0) \vee (NB\_AW = 0)$. Combining these two relations we obtain the following invariant which characterizes the readers-writers problem:

$$(NB\_AR \times NB\_AW = 0) \wedge (NB\_AW \leq 1).$$

Hence, any invocation of read_file() or write_file() can be allowed only if its execution does not entail the violation of the previous relation.

Strong priority to the readers The monitor described in Fig. 3.23 provides the readers with strong priority. This means that, when a reader is reading, the readers which arrive can immediately proceed to read and, when a writer terminates writing, if readers are waiting they can immediately proceed to read.

When a reader p invokes begin_read(), it first increases NB_WR (line 1) Then, if a writer is writing, p has to wait on the condition $C\_READERS$ (line 2). As $C\_READERS$.signal() reactivates a single process, when p reactivates, it will have to reactivate another waiting reader (if any), which in turn will reactivate another one, etc., until all the readers which are waiting have been reactivated. After it was reactivated, or immediately if $NB\_AR = 0$, p updates the control registers NB_WR and NB_AR (line 3) before exiting the monitor and going to read the file.

When a reader p invokes end_read(), it first decreases NB_AR (line 5). If they are no more readers interested in reading the file, p reactivates the first writer which is waiting (if any) (line 6) and finally exits the monitor.

When a writer p invokes begin_write(), it waits (line 8) if another writer is writing ($NB\_AR \neq 0$) or readers are interested in the file ($NB\_WR + NB\_AR \neq 0$). When it is reactivated, or immediately if it does not have to wait, p increases NB_AR to indicate there is a writer accessing the file, and exits the monitor.

Finally, when a writer p invokes end_write(), it first decreases NB_AW (line 11). Then, if readers are waiting ($NB\_WR > 0$), it reactivates the first of them (which in turn will reactivate another one, etc., as seen at line 2). If no reader is waiting and at least one writer is waiting, the terminating writer reactivates another writer (line 12).

Theorem 13 *The monitor described in Fig. 3.23 solves the readers-writers problem with strong priority to the readers.*

Proof Let us first show that the predicate $(NB\_AR \times NB\_AW = 0) \wedge (NB\_AW \leq 1)$ remains always true. Let us first observe that it is initially satisfied.

Due to line 9 and the waiting predicate $NB\_AW \neq 0$ used at line 8 and line 11, it follows that the only values taken by NB_AW are 0 and 1.

```
monitor $RW\_READERS\_STRONG\_PRIORITY$ is
    $NB\_WR$, $NB\_AR$, $NB\_AW$ init 0;
    $C\_READERS$, $C\_WRITERS$: condition.

operation begin_read() is
    (1)  $NB\_WR \leftarrow NB\_WR + 1$;
    (2)  if $(NB\_AW \neq 0)$ then $C\_READERS$.wait(); $C\_READERS$.signal() end if;
    (3)  $NB\_WR \leftarrow NB\_WR - 1$; $NB\_AR \leftarrow NB\_AR + 1$;
    (4)  return()
  end operation.

operation end_read() is
    (5)  $NB\_AR \leftarrow NB\_AR - 1$;
    (6)  if $(NB\_WR + NB\_AR = 0)$ then $C\_WRITERS$.signal() end if;
    (7)  return()
end operation.

operation begin_write() is
    (8)  if $\big((NB\_AW \neq 0) \vee (NB\_WR + NB\_AR \neq 0)\big)$ then $C\_WRITERS$.wait() end if;
    (9)  $NB\_AW \leftarrow NB\_AW + 1$;
    (10) return()
end operation.

operation end_write() is
    (11) $NB\_AW \leftarrow NB\_AW - 1$;
    (12) if $(NB\_WR > 0)$ then $C\_READERS$.signal() else $C\_WRITERS$.signal() end if;
    (13) return()
end operation.
end monitor.
```

Fig. 3.23 A readers-writers monitor with strong priority to the readers

Let us now assume that *NB_AW* $= 1$. Then, due to the transfer of predicate (between lines 6 and 8 or between lines 12 and 8) we have *NB_AR* $+$ *NB_AW* $= 0$, from which we conclude *NB_AR* $= 0$, and consequently *NB_AR* $\times$ *NB_AW* $= 0$.

Let us now assume that *NB_AR* > 0. This register is increased at line 3. Due to the waiting predicate *NB_AW* > 0 used at line 2 and the transfer of predicate between line 12 (where we also have *NB_AW* $= 0$) and line 2, it follows that *NB_AW* $= 0$ when line 3 is executed. Consequently, we have (*NB_AR* > 0) $\Rightarrow$ (*NB_AW* $= 0$), which completes the proof of the safety property of the readers-writers problem.

Let us now prove the liveness property, namely strong priority to the readers. Let us first observe that, if a read is allowed to read, we have *NB_AR* > 0 and, consequently, *NB_AW* $= 0$. It then follows from the waiting predicate used at line 2 that all the readers which invoke begin_read() are allowed to read.

Let us now consider that readers invoke begin_read() while a writer has previously been allowed to write. We then have *NB_AW* > 0 (line 9 executed by the writer) and *NB_AR* > 0 (line 1 executed later by readers). It follows that, when the writer invokes end_write(), it will execute *C_READERS*.up() (line 12) and reactivate a reader (which in turn will reactivate another reader, etc.). Consequently, when a writer is writing and there are waiting readers, those readers proceed to read when the writer terminates, which concludes the proof of the liveness property. □

Strong priority to the writers The monitor described in Fig. 3.24 provides strong priority to the writers. This means that, as soon as writers want to write, no more readers are allowed to read until they have all terminated. The text of the monitor is self-explanatory.

When looking at Fig. 3.24, as far as the management of priority is concerned, it is important to insist on the role played by the register *NB_WW*. This register stores the actual number of processes which want to write and are blocked. Hence, giving strong priority to the writers is based on the testing of that register at line 1 and line 12. Moreover, when the priority is given to the writers, the register *NB_WR* (which counts the number of waiting readers) is useless.

Similarly, the same occurs in the monitor described in Fig. 3.23. Strong priority is given to the readers with the help of the register *NB_WR* while, in that case, the register *NB_WW* becomes useless.

A type of fairness Let us construct a monitor in which, while all invocations of conc_read_file() and conc_write_file() (as defined in Fig. 3.10) terminate, the following two additional liveness properties are satisfied:

```
monitor $RW\_WRITERS\_STRONG\_PRIORITY$ is
    $NB\_AR, NB\_WW, NB\_AW$ init 0;
    $C\_READERS, C\_WRITERS$: condition.

operation begin_read() is
   (1)  if $(NB\_WW + NB\_AW \neq 0)$ then $C\_READERS$.wait(); $C\_READERS$.signal() end if;
   (2)  $NB\_AR \leftarrow NB\_AR + 1$;
   (3)  return()
  end operation.

operation end_read() is
   (4)  $NB\_AR \leftarrow NB\_AR - 1$;
   (5)  if $(NB\_AR = 0)$ then $C\_WRITERS$.signal() end if;
   (6)  return()
end operation.

operation begin_write() is
   (7)  $NB\_WW \leftarrow NB\_WW + 1$;
   (8)  if $\big((NB\_AW \neq 0) \vee (NB\_AR \neq 0)\big)$ then $C\_WRITERS$.wait() end if;
   (9)  $NB\_WW \leftarrow NB\_WW - 1$; $NB\_AW \leftarrow NB\_AW + 1$;
   (10) return()
end operation.

operation end_write() is
   (11) $NB\_AW \leftarrow NB\_AW - 1$;
   (12) if $(NB\_WW > 0)$ then $C\_WRITERS$.signal() else $C\_READERS$.signal() end if;
   (13) return()
end operation.
end monitor.
```

Fig. 3.24 A readers-writers monitor with strong priority to the writers

- Property P1: When a write terminates, all waiting readers are allowed to read before the next write.
- Property P2: When there are readers which are reading the file, the newly arriving readers have to wait if writers are waiting.

These properties are illustrated in Fig. 3.25, where indexes are used to distinguish different executions of a same operation. During an execution of conc_write_file$_1$(), two readers invoke conc_read_file$_1$() and conc_read_file$_2$() and a writer invokes conc_write_file$_2$(). As there is currently a write on the file, these operations are blocked inside the monitor (to preserve the monitor invariant). When conc_write_file$_1$() terminates, due to property P1, the invocations conc_read_file$_1$() and conc_read_file$_2$() are executed. Then, while they are reading the file, conc_read_file$_3$() is invoked. Due to property P2, this invocation must be blocked because, despite the fact that the file is currently being read, there is a write waiting. When conc_read_file$_1$() and conc_read_file$_2$() have terminated, conc_write_file$_2$() can be executed. When this write terminates, conc_read_file$_3$() and conc_read_file$_4$() are executed. Etc.

The corresponding monitor is described in Fig. 3.26. The difference from the previous readers-writers monitors lies in the way the properties P1 and P2 are ensured. The property P2 is ensured by the waiting predicate at line 2. If a writer is writing or waiting (predicate ($NB\_WW + NB\_AW \neq 0$)) when a reader arrives, the reader has to wait, and when this reader reactivates it will propagate the reactivation to another waiting reader (if any) before starting to read. The property P1 is ensured by the reactivating predicate ($NB\_WR > 0$) used at line 13: if there are readers that are waiting when a writer terminates, the first of them is reactivated, which reactivates the following one (statement *C_READERS*.signal() at line 2), etc., until all waiting readers have been reactivated.

The reader can check that the implementation of such fairness properties would have been much more difficult if one was asked to implement them directly from semaphores. ("Directly" meaning here: without using the translation described in Fig. 3.22.)

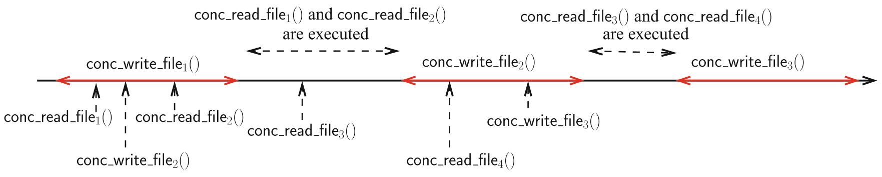


Fig. 3.25 The fairness properties P1 and P2

```
monitor $RW\_WRITERS\_STRONG\_PRIORITY$ is
    $NB\_WR, NB\_AR, NB\_WW, NB\_AW$ init 0;
    $C\_READERS, C\_WRITERS$: condition.

operation begin_read() is
   (1)  $NB\_WR \leftarrow NB\_WR + 1$;
   (2)  if $(NB\_WW + NB\_AW \neq 0)$ then $C\_READERS$.wait(); $C\_READERS$.signal() end if;
   (3)  $NB\_WR \leftarrow NB\_WR - 1$; $NB\_AR \leftarrow NB\_AR + 1$;
   (4)  return()
  end operation.

operation end_read() is
   (5)  $NB\_AR \leftarrow NB\_AR - 1$;
   (6)  if $(NB\_AR = 0)$ then $C\_WRITERS$.signal() end if;
   (7)  return()
end operation.

operation begin_write() is
   (8)  $NB\_WW \leftarrow NB\_WW + 1$;
   (9)  if $\big((NB\_AW \neq 0) \vee (NB\_AR \neq 0)\big)$ then $C\_WRITERS$.wait() end if;
   (10) $NB\_WW \leftarrow NB\_WW - 1$; $NB\_AW \leftarrow NB\_AW + 1$;
   (11) return()
end operation.

operation end_write() is
   (12) $NB\_AW \leftarrow NB\_AW - 1$;
   (13) if $(NB\_WR > 0)$ then $C\_READERS$.signal() else $C\_WRITERS$.signal() end if;
   (14) return()
end operation.
end monitor.
```

Fig. 3.26 A readers-writers monitor with fairness properties

3.3.6 Scheduled Wait Operation

Parameterized wait operation Variants of the monitor concept provide conditions C (internal queues) with a parameterized operation C.wait(x). The parameter x is a positive integer defining a priority. The smaller the value of x, the higher the priority for the corresponding process to be reactivated. (When there is no parameter x, the processes are reactivated in the order in which they invoked C.wait().)

An activation-on-deadline monitor As a simple example of use of the parameterized wait operation, let us consider a monitor which provides the processes with an operation denoted C.wake_up(x) which allows the invoking process to be woken up x units of time after its invocation time.

The corresponding monitor is described in Fig. 3.27. When a process p invokes wake_up_at(x), it computes its wake up date (line 1) and adds it to into a bag denoted *BAG* (line 2). A *bag* is a set that can contain the same element several times (a bag is also called a *multiset*). The statement "add d to *BAG*" adds one copy of d to the bag,

```
monitor $WAKE\_UP$ is
    $BAG$ init $\emptyset$; $CLOCK$ init 0;
    $QUEUE$: condition.

operation wake_up_at($x$) is
    (1)  $wake\_up\_date \leftarrow CLOCK + x$;
    (2)  add $wake\_up\_date$ to $BAG$;
    (3)  $QUEUE$.wait($wake\_up\_date$);
    (4)  suppress $wake\_up\_date$ from $BAG$;
    (5)  return()
end operation.

operation tic() is
    (6)  $CLOCK \leftarrow CLOCK + 1$;
    (7)  $now \leftarrow CLOCK$;
    (8)  while ($now \in BAG$) do $QUEUE$.signal() end while;
    (9)  return()
end operation.
end monitor.
```

Fig. 3.27 A monitor based on a scheduled wait operation

while the statement "*d* from *BAG*" removes one copy of *d* from the bag (if any). Then, *p* invokes *QUEUE*.wait(*wake_up_date*) (line 3). When it is reactivated, *p* removes its wake up date from the bag (line 4).

The second operation, denoted tic(), is executed by the system at the end of each time unit (it can also be executed by a specific process whose job is to measure the physical or logical passage of time). This operation first increases the monitor clock *CLOCK* (line 6). Then, it reactivates, one after the other, all the processes whose wake up time is equal to *now* (the current time value) (lines 7–8).

3.4 Declarative Synchronization: Path Expressions

The monitor concept allows concurrent objects to be built by providing (a) sequentiality inside a monitor and (b) condition objects to solve internal synchronization issues. Hence, as we have seen, monitor-based synchronization is fundamentally an imperative approach to synchronization. This section shows that, similarly to sequential programming languages, the statement of synchronization can be imperative or declarative.

As monitors, several path expression formulations have been introduced. We considered here the one that was introduced in a variant of the Pascal programming language.

3.4.1 Definition

The idea of path expressions is to state constraints on the order in which the operations on a concurrent object have to be executed. To that end, four base operators are used, namely *concurrency, sequentiality, restriction*, and *de-restriction*. It is then up to the compiler to generate the appropriate control code so that these constraints are always satisfied.

Let us consider an object defined by a set of operations. A path expression associated with this object has the form

path *path_expression* **end path**

where *path_expression* is defined as follows. The identifier of any of the object operations is a base path expression. Path expressions are then defined recursively as described in the items that follow. Let pe_1 and pe_2 be two path expressions.

- Concurrency operator (denoted ","). The statement "pe_1, pe_2" defines a path expression which imposes no restriction on the order in which pe_1 and pe_2 are executed.
- Sequentiality operator (denoted ";"). The statement "$pe_1; pe_2$" defines a path expression which states that pe_1 has to be executed before pe_2. There can be any number of concurrent executions of pe_1 and pe_2 as long as the number of executions of pe_2 that have started remains less than or equal to the number of executions of pe_2 that have terminated.
- Restriction operator (denoted "k: " where k is a positive integer). The statement "$k : pe_1$" defines a path expression which states that at most k executions of pe_1 are allowed to proceed concurrently.
- De-restriction operator (denoted "[ ]"). The statement "$[pe_1]$" defines a path expression which states that any number of executions of pe_1 are allowed to proceed concurrently.

As with arithmetic or algebraic expressions, parentheses can be used to express precedence when combining these operators to define powerful path expressions. Sequentiality and concurrency have priority over restriction and de-restriction.

Simple examples The aim of the examples that follow is to show how path expressions can be used to associate specific concurrency constraints with an object. The object that is considered has two operations denoted op_1 and op_2.

- **path** $(1 : \mathsf{op}_1),\ \mathsf{op}_2$ **end path** states that, at any time, at most one execution at a time of op_1 is allowed (hence all executions of op_1 will be sequential), while there is no constraint on the executions of op_2.
- **path** $(1 : \mathsf{op}_1),\ (1 : \mathsf{op}_2)$ **end path** states that, at any time, (a) the executions of op_1 have to be sequential, (b) the executions of op_2 have to be sequential, and (c) there is no constraint relating the executions of op_1 and op_2. It follows from this

path expression that, at any time, there is at most one execution of op_1 and one execution of op_2 which can proceed concurrently.

- **path** $2 : (\mathsf{op}_1;\ \mathsf{op}_2)$ **end path** states that, at any time, (a) the number of executions of op_2 that have started never surpasses the number of executions of op_1 that have completed (this is due to the ";" internal operator), and (b) the number of executions of op_1 that have started never surpasses by more than two the number of executions of op_2 that have completed (this is due to the "$2 : ()$" operator).
- **path** $1 : \big([\mathsf{op}_1],\ \mathsf{op}_2\big)$ **end path** states that, at any time, there is at most either one execution of op_2 or any number of concurrent executions of op_1.
- **path** $4 : \big((3 : \mathsf{op}_1),\ (2 : \mathsf{op}_2)\big)$ **end path** states that, at any time, there are at most three concurrent executions of op_1 and at most two concurrent executions of op_2, and at most four concurrent executions when adding the executions of op_1 and the executions of op_2.

3.4.2 Using Path Expressions to Solve Synchronization Problems

Readers-writers Let us consider a reader-writer sequential object defined by the operations read_file() and write_file() introduced in Sect. 3.2.4.

The following path expression $path_1$ adds control on the operation executions such that the file can now be accessed by any number of readers and any number of writers. Hence, with such a control enrichment, the file becomes a concurrent file. The path expression $path_1$ is defined as follows:

$$path_1 = \textbf{path } 1 : \big([\mathsf{read\_file}],\ \mathsf{write\_file}\big) \textbf{ end path}.$$

It is easy to see that $path_1$ states that, at any time, access to the file is restricted to a single writer or (due to the de-restriction on read_file) to any number of readers. Thus, $path_1$ gives weak priority to the readers.

Let us now replace the operation write_file() by a new write operation denoted WRITE_file() defined as follows: **operation** WRITE_file(v) **is** write_file(v); return() **end operation**. Thus, the file object offers the operations read_file() and WRITE_file() to the processes; write_file() is now an internal procedure used to implement the object operation WRITE_file(). (If the file was used by a single process, WRITE_file() and write_file() would be synonyms, which is no longer the case in a concurrency context as we are about to see.)

Considering such a file object, let us define the two following path expressions $path_2$ and $path_3$:

$$path_2 = \textbf{path } 1 : \big([\mathsf{read\_file}],\ [\mathsf{WRITE\_file}]\big) \textbf{ end path},$$
$$path_3 = \textbf{path } 1 : \mathsf{write\_file} \textbf{ end path}.$$

The path expression $path_3$ states that no two processes can write the file simultaneously. Due to its restriction operator, $path_2$ states that, at any time, access to the

```
operation $B$.prod($v$) is
    $BUF[in]$.write($v$); $in \leftarrow (in+1) \mod k$;
    return()
end operation.

operation $B$.cons() is
    $r \leftarrow BUF[out]$.read(); $out \leftarrow (out+1) \mod k$;
    return($r$)
end operation.
```

Fig. 3.28 A buffer for a single producer and a single consumer

file is given either to any number of readers (which have invoked read_file()) or any number of writers (which have invoked WRITE_file()). Moreover, $path_2$ defines a kind of alternating priority. If a reader is reading, it gives access to the file to all the readers that arrive while it is reading and, similarly, if a writer is writing, it reserves the file for all the writers that are waiting.

Producer-consumer Let us consider a buffer of size k shared by a single producer and a single consumer. Using the same base objects ($BUF[0..(k-1)]$, in and out) as in Fig. 3.4 (Sect. 3.2.2), the operations of such a buffer object B are defined in Fig. 3.28.

The following path expression $path_4$ defines the synchronization control associated with such a buffer:

$$path_4 = \textbf{path}\ k : \big(\mathsf{prod}; \mathsf{cons}\big)\ \textbf{end path}.$$

If there are both several readers and several writers, it is possible to use the same object B. (The only difference for B is that now in is shared by the producers and out is shared by the consumers, but this does not entail a modification of the code of B.) The only modification is the addition of synchronization constraints specifying that at most one producer at a time is allowed to produce and at most one consumer at a time is allowed to consume.

Said differently, the only modification is the replacement of $path_4$ by $path_5$, defined as follows:

$$path_5 = \textbf{path}\ k : \big((1 : \mathsf{prod}); ((1 : \mathsf{cons})\big)\ \textbf{end path}.$$

3.4.3 A Semaphore-Based Implementation of Path Expressions

This section presents a semaphore-based implementation of path expressions. An implementation pattern is defined for each operator (concurrency, sequentiality, restriction, and de-restriction). For each object operation op(), we obtain a control prefix and a control suffix that are used to bracket any invocation of the operation. These prefixes and suffixes are the equivalent of the control operations begin_op() and end_op() which have to be explicitly defined when using an imperative approach.

Generating prefixes and suffixes Let pe denote a path expression. $\langle$prefix(pe)$\rangle$ and $\langle$suffix(pe)$\rangle$ denote the code prefix and the code suffix currently associated with pe. These prefixes and suffixes are defined recursively starting with the path expression pe and proceeding until the prefix and suffix of each operation is determined. Initially, $\langle$prefix(pe)$\rangle$ and $\langle$suffix(pe)$\rangle$ are empty control sequences.

1. Concurrency rule. Let $pe = pe_1, pe_2$. The expression $\langle$prefix(pe)$\rangle$ pe_1, pe_2 $\langle$suffix(pe)$\rangle$ gives rise to the two expressions $\langle$prefix(pe)$\rangle$ pe_1 $\langle$suffix(pe)$\rangle$ and $\langle$prefix(pe)$\rangle$ pe_2 $\langle$suffix(pe)$\rangle$, which are then considered separately.
2. Sequentiality rule. Let $pe = pe_1; pe_2$. The expression $\langle$prefix(pe)$\rangle$ pe_1; $pe_2\langle$suffix(pe)$\rangle$ gives rise to two expressions which are then considered separately, namely the expression $\langle$prefix(pe)$\rangle$ pe_1 $\langle S$.up()$\rangle$ and the expression $\langle S$.down()$\rangle$ $pe_2\langle$suffix(pe)$\rangle$ where S is a new semaphore initialized to 0. As we can see, the aim of the semaphore S is to force pe_2 to wait until pe_1 is executed.

   Hence, for the next step, as far as pe_1 is concerned, we have prefix(pe_1) = prefix(pe) and suffix(pe_1)=S.up(). Similarly, we have prefix(pe_2)=S.down() and suffix(pe_1) = S.up().
3. Restriction rule. Let $pe = k : pe_1$. The expression $\langle$prefix(pe)$\rangle k : pe_1\langle$suffix(pe)$\rangle$ gives rise to the expression $\langle S'$.down(); prefix(pe)$\rangle$ pe_1 $\langle$suffix(pe); S'.up()$\rangle$, where S' is a new semaphore initialized to k.

   Hence, we have prefix(pe_1) = S'.down(); prefix(pe) and suffix(pe_1) = suffix(pe); S'.up() to proceed recursively (if pe_1 is not an operation name).
4. De-restriction rule. Let pe = $[pe_1]$. The expression $\langle$prefix(pe)$\rangle[pe_1]$ $\langle$suffix(pe)$\rangle$ gives rise to the expression $\langle$prio_down(CT, S'', prefix(pe))$\rangle$ pe_1 $\langle$prio_up(CT, S'', suffix(pe))$\rangle$, where
   - CT is a counter initialized to 0,
   - S'' is a new semaphore initialized to 1, and
   - prio_down() and prio_up() are defined as indicated in Fig. 3.29.

```
operation prio_up($CT, S''$, suffix($pe$)) is
        $S''$.down(); $CT \leftarrow CT + 1$;
        if ($CT = 1$) then suffix($pe$) end if;
        $S''$.up(); return()
end operation.

operation prio_down($CT, S''$, prefix($pe$)) is
        $S''$.down(); $CT \leftarrow CT - 1$;
        if ($CT = 0$) then prefix($pe$) end if;
        $S''$.up(); return()
end operation.
```

Fig. 3.29 Operations prio_down() and prio_up()

The aim of these operations is to give priority to the processes that invoke an operation involved in the path expression pe_1. (The reader can check that their internal statements are the same as the ones used in Fig. 3.11 to give weak priority to the readers.)

An example To illustrate the previous rules, let us consider the following path expression involving three operations denoted op_1, op_2, and op_3:

$$\textbf{path } 1 : \big([\mathsf{op}_1; \mathsf{op}_2],\ \mathsf{op}_3\big) \textbf{ end path}.$$

Hence, we have initially $pe = 1 : \big([\mathsf{op}_1; \mathsf{op}_2],\ \mathsf{op}_3\big)$, $\mathsf{prefix}(pe) = \mathsf{suffix}(pe) = \epsilon$ (where ϵ represents the empty sequence). Let us now apply the rules as defined by their precedence order. This is also described in Fig. 3.30 with the help of the syntactical tree associated with the considered path expression.

- Let us first apply the restriction rule (item 3). We obtain $k = 1$ with $pe_1 = [\mathsf{op}_1; \mathsf{op}_2],\ \mathsf{op}_3$. It follows from that rule that $\mathsf{prefix}(pe_1) = S1.\mathsf{down}(); \epsilon$ and $\mathsf{suffix}(pe_1) = \epsilon; S1.\mathsf{up}()$, where $S1$ is a semaphore initialed to 1.
- Let us now apply the concurrency rule (item 1). We have $pe_1 = pe_2, pe_3$, where $pe_2 = [\mathsf{op}_1; \mathsf{op}_2]$ and $pe_3 = \mathsf{op}_3$. It follows from that rule that:
  - $\mathsf{prefix}(\mathsf{op}_3) = \mathsf{prefix}(pe_1) = S1.\mathsf{down}()$ and $\mathsf{suffix}(\mathsf{op}_3) = \mathsf{suffix}(pe_1) = S1.\mathsf{up}()$. Hence, any invocation of $\mathsf{op}_3()$ has to be bracketed by $S1.\mathsf{down}()$ and $S1.\mathsf{up}()$.
  - Similarly, $\mathsf{prefix}(pe_2) = \mathsf{prefix}(pe_1) = S1.\mathsf{down}()$ and $\mathsf{suffix}(pe_2) = \mathsf{suffix}(pe_1) = S1.\mathsf{up}()$.
- Let us now consider $pe_2 = [\mathsf{op}_1; \mathsf{op}_2] = [pe_4]$. Applying the de-restriction rule (item 4) we obtain $\mathsf{prefix}(pe_4) = \mathsf{prio\_down}(CT, S2, \mathsf{prefix}(pe_2))$ and $\mathsf{suffix}(pe_4)$

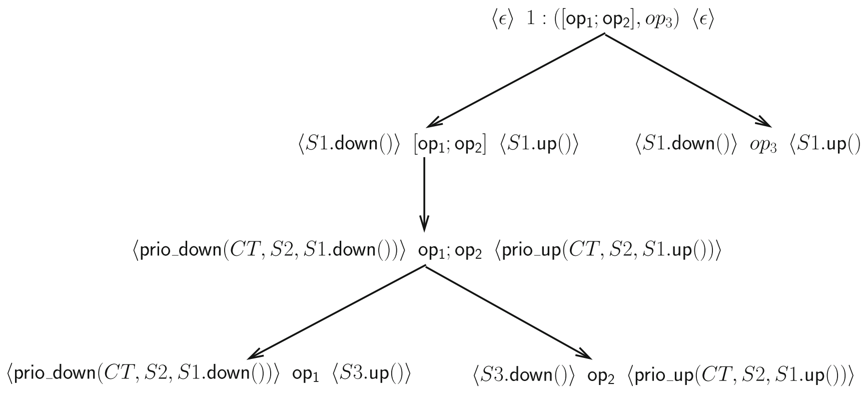

⟨ϵ⟩ 1 : ([op1; op2], op3) ⟨ϵ⟩
⟨S1.down()⟩ [op1; op2] ⟨S1.up()⟩
⟨S1.down()⟩ op3 ⟨S1.up()⟩
⟨prio_down(CT, S2, S1.down())⟩ op1; op2 ⟨prio_up(CT, S2, S1.up())⟩
⟨prio_down(CT, S2, S1.down())⟩ op1 ⟨S3.up()⟩
⟨S3.down()⟩ op2 ⟨prio_up(CT, S2, S1.up())⟩

Fig. 3.30 Derivation tree for a path expression

```
operation start_op1() is
        $S2$.down();
        $CT \leftarrow CT + 1$; if ($CT = 1$) then $S1$.down() end if;
        $S2$.up(); return()
end operation.

operation end_op1() is $S3$.up(); return() end operation.

operation start_op2() is $S3$.down(); return() end operation.

operation end_op2() is
        $S2$.down();
        $CT \leftarrow CT + 1$; if ($CT = 0$) then $S1$.up() end if;
        $S2$.up(); return()
end operation.

operation start_op3() is $S1$.down(); return() end operation.

operation end_op3() is $S1$.up(); return() end operation.
```

Fig. 3.31 Control prefixes and suffixes automatically generated

= prio_up(CT, $S2$, suffix(pe_2)), where CT is a counter initialized to 0 and $S2$ a semaphore initialized to 1, i.e.,

– prefix(op$_1$; op$_2$) = prio_down(CT, $S2$, $S1$.down()), and
– suffix(op$_1$; op$_2$) = prio_up(CT, $S2$, $S1$.up()).

- Finally, let us apply the sequentiality rule (item 2) to pe_4 = op$_1$; op$_2$. A new semaphore $S3$ initialized to 0 is added, and we obtain

  – prefix(op$_1$) = prio_down(CT, $S2$, $S1$.down()),
  – suffix(op$_1$) = $S3$.up(),
  – prefix(op$_2$) = $S3$.down(), and
  – suffix(op$_2$) = prio_up(CT, $S2$, $S1$.up()).

These prefixes and suffixes, which are automatically derived from the path expression, are summarized in Fig. 3.31, where the code of prio_down and prio_up() is explicit.

3.5 Summary

This chapter has presented the semaphore object and two programming language constructs (monitors and path expressions) that allow the design of lock-based atomic objects. Such language constructs provide a higher abstraction level than semaphores or base mutual exclusion when one has to reason on the implementation of concurrent objects. Hence, they can make easier the job of programmers who have to implement concurrent objects.

3.6 Bibliographic Notes

- The concept of a semaphore was introduced by E.W. Dijkstra in [89, 90].

  Several variants of semaphores have since been proposed. Lots of semaphore-based solutions to synchronization problems are described in [91].
- The readers-writers problem was defined by P.J. Courtois, F. Heymans, and D.L. Parnas [80], who also presented semaphore-based solutions.

  The use buffers to reduce delays for readers and writers is from [44].
- The concept of a monitor is due to P. Brinch Hansen [56, 57] and C.A.R. Hoare [150]. This concept originated from the "secretary" idea of E.W. Dijkstra [89].
- The notion of transfer of predicate is from [48].
- An alternative to monitor conditions (queues) can be found in [174].
- A methodology for proving monitors is presented in [63]. Proof methodologies for concurrent objects can be found in [27, 49, 69, 220].
- Path expressions were introduced by R.H. Campbell and N. Haberman [63]. An implementation is described in [64]. Extensions are proposed in [62].
- Other formalisms have been proposed to express synchronization on concurrent objects. Counters [116, 246] and serializers [149] are among the most well known.

3.7 Exercises and Problems

1. Considering the implementation of a semaphore S, prove the invariant that relates the number of processes which are blocked on S and the value of the implementation counter associated with S.
2. Implement a rendezvous object from semaphores. This object must be a multi-shot object which means that the same object can be repeatedly used by the same set of processes (e.g., to re-synchronize at the beginning of a parallel loop).
3. Design two algorithms which implement the FIFO access rule for the readers-writers problem: one based on semaphores, the second on a monitor. "FIFO access rule" means that, in addition to the exclusion rules of the readers-writers problem, the readers and the writers have to access in their arrival order, namely:

```
operation $B$.produce($v$) is
        $FREE$.down();
        $MP$.down(); $my\_index \leftarrow IN$; $IN \leftarrow (IN + 1) \mod k$; $MP$.up();
        $BUF[my\_index]$.write($v$);
        $BUSY$.up(); return()
end operation.

operation $B$.consume() is
        $BUSY$.down();
        $MC$.down(); $my\_index \leftarrow OUT$; $OUT \leftarrow (OUT + 1) \mod k$; $MC$.up();
        $r \leftarrow BUF[my\_index]$.read();
        $FREE$.up(); return($r$)
end operation.
```

Fig. 3.32 A variant of a semaphore-based implementation of a buffer

- A reader that arrives while another reader is reading can immediately access the file if no writer is waiting. Otherwise, the reader has to wait until the writers that arrived before it have accessed the file.
- A writer cannot bypass the writers and the readers which arrived before it.

4. Consider the producers-consumers algorithm described in Fig. 3.32, which is a variant of the solution given in Fig. 3.4.

   The semaphores *FREE* and *BUSY* have the same meaning as in Fig. 3.4. *FREE* (which counts the number of available entries) is initialized to k, while *BUSY* (which counts the number of full entries) is initialized to 0. *MP* (or *MC*) which is initialized to 1, is used to prevent producers (or consumers) from simultaneously accessing *IN* (or *OUT*).

   Is this algorithm correct? (To show that it is correct, a proof has to be given. To show that it is incorrect, a counter-example has to be exhibited.)

5. For a single producer and a single consumer, let us consider the two solutions described in Fig. 3.33. These solutions are based neither on semaphores nor on monitors.

   The integers *IN* and *OUT* are SWMR atomic registers initialized to 0, and the array $BUF[0..(k-1)]$ (with $k \geq 2$) is also made up of SWMR atomic registers. *IN* and $BUF[0..(k-1)]$ are written only by the producer, while *OUT* is written only by the consumer.

   These two algorithms differ only in the management of the counter indexes *IN* and *OUT*. In solution 1, their domain is bounded, namely $[0..k-1]$, while it is not in solution 2.

   - Let $k \geq 2$. Prove that both solutions are correct.
   - Let $k = 1$. Is solution 1 correct? Is solution 2 correct?

```
=================== Solution # 1 ===============
operation $B$.produce($v$) is
    wait $\big((IN+1) \mod k) \neq OUT\big)$;
    $BUF[IN]$.write($v$); $IN \leftarrow (IN+1) \mod k$;
    return()
end operation.

operation $B$.consume() is
    wait $\big(IN \neq OUT\big)$;
    $r \leftarrow BUF[OUT]$.read(); $OUT \leftarrow (OUT+1) \mod k$;
    return($r$)
end operation.

=================== Solution # 2 ===============
operation $B$.produce($v$) is
    wait $\big()(IN \neq (OUT+k)\big)$;
    $BUF[IN \mod k]$.write($v$); $IN \leftarrow IN+1$;
    return()
end operation.

operation $B$.consume() is
    wait $\big(OUT \neq IN\big)$;
    $r \leftarrow BUF[OUT \mod k]$.read(); $OUT \leftarrow OUT+1$;
    return($r$)
end operation.
```

Fig. 3.33 Two buffer implementations

- Considering each solution, is it possible that all entries of $BUF[1..(k-1)]$ contain item values produced and not yet consumed?
- What tradeoff exists between (a) the fact that IN and OUT are bounded and (b) the fact that all entries of $BUF[1..(k-1)]$ can be simultaneously full?

6. The algorithm in Fig. 3.34 describes a solution to the readers-writers problem. This solution is based on a semaphore, denoted *MUTEX* and initialized to 1, and an array *FLAG*[1..*n*] of SWMR atomic registers with one entry per process p_i. The atomic register *FLAG*[*i*] is set to *true* by p_i when it wants to read the file and reset to *false* after it has read it. *MUTEX* is a mutex semaphore that allows a writer to exclude other writers and all readers.

- Prove that the algorithm is correct (a writer executes in mutual exclusion and readers are allowed to proceed concurrently).
- What type of priority is offered by this algorithm?
- In the worst case how many process can be blocked on the **wait** statement?
- Let us replace the array *FLAG*[1..*n*] of SWMR atomic registers by a single MWMR atomic register *READERS* initialized to 0, and

```
operation begin_read() is
        $MUTEX$.down(); $FLAG[i] \leftarrow true$; $MUTEX$.up(); return()
end operation.

operation end_read() is $FLAG[i] \leftarrow false$; return() end operation.

operation begin_write() is
        $MUTEX$.down(); wait $\big(\forall\, i : (\neg FLAG[i])\big)$; return()
end operation.

operation end_write() is $MUTEX$.up(); return() end operation.
```

Fig. 3.34 A readers-writers implementation

- The statement $FLAG[i] \leftarrow true$ is replaced by $READERS \leftarrow READERS + 1$,
- The statement $FLAG[i] \leftarrow false$ is replaced by $READERS \leftarrow READERS - 1$, and
- The predicate $\forall\, i : (\neg FLAG[i])$ is replaced by $READERS = 0$.

Is this modified solution correct? Explain why?

7. Design an efficient monitor-based solution to the producers-consumers problem based on the Boolean arrays $FULL[1..n]$ and $EMPTY[1..n]$ used in Fig. 3.7. As for the semaphore-based algorithm described in this figure, "efficient" means here that producers must be allowed to produce concurrently and consumers must be allowed to consume concurrently.

8. In a lot of cases, the invocation of the C.signal() operation of a monitor appears as the last invocation inside a monitor operation. Design an efficient implementation of a monitor that takes into account this feature.

9. Let us associate the semantics "signal all" with the C.signal() operation on each condition C of a monitor. This semantics means that (if any) all the processes which are blocked on the condition C are reactivated and have priority to obtain mutual exclusion and re-access the monitor. The process which has invoked C.signal() continues its execution inside the monitor. Considering this "signal all" semantics:

   - Design a readers-writers monitor with strong priority to the writers.
   - Design an implementation for "signal all" monitors from underlying semaphores.

10. The last writer. Let us consider the monitor-based solution with strong priority to the readers (Fig. 3.23). Modify this solution so that only the last writer can be blocked (it can be blocked only because a reader is reading or a writer is writing). This means that, when a writer p invokes begin_write(), it unblocks the waiting

writer q if there is one. The write (not yet done) of q is then "overwritten" by the write of p and the invocation of begin_write() issued by q returns *false*.

To that end, the operation s conc_write_file(v) defined in Fig. 3.10 is redefined as follows:

```
operation conc_write_file($v$) is
    $r \leftarrow$ begin_write();
    if ($r$) then write_file($v$); end_write() end if;
    return($r$)
end operation.
```

Design an algorithm implementing the corresponding begin_write() operation. This operation returns a Boolean whose value is *true* if the write of the value v has to be executed.

11. Implement semaphores (a) from monitors, and (b) from path expressions.

12. Let us consider the railways system described in Fig. 3.35.

- Any number of trains can go concurrently from A to B, or from B to A, but not at the same time (a single railway has to be shared and can be used in one direction only at any time).
- The same between D and C.
- There is a unidirectional railway from B to C (upper arrow) but it can be used by only one train at a time.
- There is a unidirectional railway from C to B (lower arrow) but it can be used by only one train at a time.

This problem is on resource allocation. It includes issues related to process (train) interactions (trains in opposite directions cannot use AB, or CD, at the same time) and issues related to the fact that some resources have a bounded capacity (mutual exclusion: at most one train at a time can go from B to C and at most one train at a time can go from C to B).

A train is going either from A to D or from D to A. A train (process) from A to D has to execute the following operations:

(a) start_from_A() when it starts from A.

(b) leave_B_to_C() when it arrives at B and tries to enter the line BC.

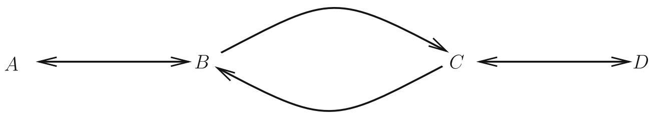

$$\begin{tikzcd} A \arrow[r, leftrightarrow] & B \arrow[r, bend left] & C \arrow[l, bend left] \arrow[r, leftrightarrow] & D \end{tikzcd}$$

Fig. 3.35 Railways example

(c) leave_C_to_D() when it arrives at C and tries to enter the line CD.

(d) arrive_in_D() when it arrives at D.

The same four operations are defined for the trains that go from D to A (A, B, C, and D are replaced by D, C, B, and A).

Design first a deadlock-free monitor that provides the processes (trains) with the previous eight operations. Design then a starvation-free monitor.

Hints.

- The internal representation of the monitor will be made of:
  - The integer variables NB_AB, NB_BA, NB_CD, and NB_DC, where NB_xy represents the number of trains currently going from x to y.

    The binary variables NB_BC and NB_CB, whose values are 0 or 1.

    All these control variables are initialized to 0.
  - The six following conditions (queues): $START\_FROM\_A$, $ENTER\_BC$, $ENTER\_CD$, $START\_FROM\_D$, $ENTER\_CB$, $ENTER\_BA$.
- The code of a process going from A to D is:

  start_from_A(); . . . ; leave_B_to_C();...; leave_C_to_D(); . . . ; arrive_in_D)().
- Before deriving the predicates that allow a train to progress when it executes a monitor operation, one may first prove that the following relation must be an invariant of the monitor internal representation:

$$\begin{aligned}
&(0 \le NB\_AB, NB\_BA, NB\_CD, NB\_DC) \wedge (0 \le NB\_BC, NB\_CB \le 1)\\
\wedge\ \ &(NB\_AB \times NB\_BA = 0) \text{ (mutual exclusion on the part } AB)\\
\wedge\ \ &(NB\_CD \times NB\_DC = 0) \text{ (mutual exclusion on the part } CD)\\
\wedge\ \ &\big((NB\_AB + NB\_BC \le 1) \vee (NB\_DC + NB\_CB \le 1)\big) \text{ (no deadlock).}
\end{aligned}$$

13. Eventcounts and sequencers.

An eventcount EC is a counting object that provides processes with two operations denoted EC.advance() and EC.wait(). EC.advance() increases the counter by 1, while EC.wait(x) blocks the invoking process until the counter is equal to or greater than x.

A sequencer SQ is an object that outputs sequence numbers. It provides the processes with the operation SQ.ticket(), which returns the next sequence number.

- Design a solution to the (one)producer-(one)consumer problem based on two eventcounts denoted IN and OUT.

```
operation conc_write_file() is
        begin_write(); write_file($v$); return()
end operation.

operation begin_write() is return() end operation.

operation conc_read_file() is
        begin_read(); $r \leftarrow$ read_file(); return($r$)
end operation.

operation begin_read() is return() end operation.
```

Fig. 3.36 Another readers-writers implementation

- Show that it is impossible to solve the producers-consumers problem using eventcounts only. (Some mutual exclusion on producers on one side and consumers on the other side is needed).
- Design a solution to the producers-consumers problem based on two eventcount and two sequencer objects.
- Implement a semaphore from a sequencer and an eventcount.
- Design a solution to the readers-writers problem from sequencers and eventcounts.
- Let an eventcount *EC* that takes values between 0 and $2^\ell - 1$ be represented by a bit array $B[1..\ell]$. Design algorithms which implement the operations *EC*.advance() and *EC*.wait() assuming that the eventcount *EC* is represented by a Gray code (which has the important property that *EC*.advance() requires only a single bit to be written).

Solutions in [243].

14. Priority with path expressions.

    Let us consider the readers-writers problem where the operations read_file() and read_file() are the base operations which access the file (see Fig. 3.10).
    In order to define an appropriate priority rule, new algorithms implementing the operations conc_read_file() and conc_write_file() are defined as described in Fig. 3.36. These implementations use the underlying operations begin_read() and begin_write(), which are pure control operations.

    Let us consider that the invocations to these operations are controlled by the following pair of path expressions:

$$path_1 = \textbf{path } 1 : \big(\text{begin\_read},\ [\text{begin\_write; write\_file}]\big) \textbf{ end path},$$
$$path_2 = \textbf{path } 1 : \big([\text{begin\_read; read\_file}],\ \text{write\_file}\big) \textbf{ end path}.$$

    Let us observe that *path*$_2$ defines mutual exclusion between a write invocation and any other operation invocation while allowing concurrent read operations.

The combination of the path expressions *path*$_1$ and *path*$_2$ defines the associated priority. What type of priority is defined?

Solutions in [63].

Part II
On the Foundations Side: The Atomicity Concept

This part of the book is made up of a single chapter that introduces the atomicity concept (also called linearizability). This concept (which was sketched in the first part of the book) is certainly (with non-determinism) one of the most important concepts related to the concurrency and synchronization of parallel and distributed programs. It is central to the understanding and the implementation of concurrent objects. This chapter presents a formal definition of atomicity and its main properties. Atomicity (which is different from sequential consistency or serializability) is the most popular consistency condition. This is due to the fact that atomic objects compose "for free".

Chapter 4
Atomicity: Formal Definition and Properties

Atomicity is a *consistency condition*, i.e., it allows us to answer the following question: Is this implementation of a concurrent object correct? The atomicity notion for read/write registers was introduced in Chap. 1, where algorithms that solve the mutual exclusion problem (i.e., algorithms which implement lock objects) were presented. Chap. 3 presented semaphore objects and programming language constructs which allow designers of concurrent objects to benefit from lock objects.

While atomicity was already informally introduced in Chap. 1, this chapter presents it from a very general and formal perspective. In the literature, the term "atomicity" is sometimes restricted to registers, while its extension to any concurrent object is usually called *linearizability*. This chapter considers both these words as synonyms.

Keywords Atomicity · Legal history · History · Linearizability · Locality property · Partial operation · Sequential consistency · Sequential history · Serializability · Total operation

4.1 Introduction

Fundamental issues Fundamental questions for a concurrent object designer are the following:

- How can the behavior of a concurrent object be specified?
- What is a correct execution of a set of processes accessing one or several concurrent objects?
- When considering object implementations that are not based on locks, how can correctness issues be addressed if one or more processes stop their execution (fail)

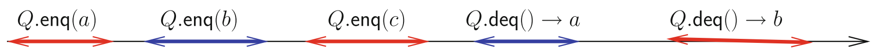


Fig. 4.1 A sequential execution of a queue object

in the middle of an operation? (This possibility was not considered in the previous chapters.)

Example To give a flavor of these questions, let us consider an unbounded first-in first-out (FIFO) queue denoted Q which provides the processes with the following two operations:

- Q.enq(v), which adds the value v at the tail of the queue, and
- Q.deq(), which returns the value at the head of the queue and suppresses it from the queue. If the queue is empty, the default value $\perp$ is returned.

Figure 4.1 describes a sequential execution of a system made up of a single process using the queue. The time line, going from left to right, describes the progress of the process when it enqueues first the value a, then the value b, and finally the value c. According to the expected semantics of a queue, and as depicted in the figure, the first invocation of Q.deq() returns the value a, the second returns the value b, etc.

Figure 4.2 depicts an execution of a system made up of two processes sharing the same queue. Now, process p_1 enqueues first a and then b whereas process p_2 concurrently enqueues c. As shown in the figure, the execution of Q.enq(c) by p_2 overlaps the executions of both Q.enq(a) and Q.enq(b) by p_1. Such an execution raises many questions, including the following: What values are dequeued by p_1 and p_2? What values can be returned by a process, say p_1, if the other process, p_2, stops forever in the middle of an operation? What happens if p_1 and p_2 share several queues instead of a single one?

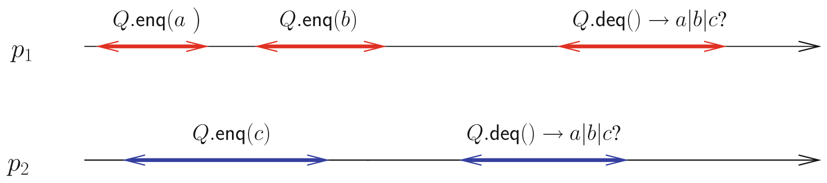

Q.enq(a)
Q.enq(b)
Q.deq() $\rightarrow a|b|c$?
p_1
Q.enq(c)
Q.deq() $\rightarrow a|b|c$?
p_2

Fig. 4.2 A concurrent execution of a queue object

4.2 Computation Model

Addressing the previous questions and related issues start from the definition of a precise computation model. This chapter presents first the base elements of such a model and the important notion of a concurrent computation *history*.

4.2.1 Processes and Operations

The computation model consists of a finite set of *n processes*, denoted $p_1, \ldots, p_n$. In order to collectively solve a given problem, the processes cooperate and synchronize their activities by accessing *concurrent objects*. The set of processes and objects is defined from a multiprocess program.

Operation execution and events Processes synchronize by executing operations exported by concurrent objects. An execution by a process of an operation on an object X is denoted $X.\mathsf{op}(arg)(res)$, where *arg* and *res* denote, respectively, the input and output parameters of the invocation. The output corresponds to the reply to the invocation. The notation $X.\mathsf{op}$ is sometimes used when the input and output parameters are not important.

When there is no ambiguity, we sometimes say *operations* where we should say *operation invocations*. The execution of an operation op() on an object X by a process p_i is modeled by two events, namely the event denoted $inv[X.\mathsf{op}(arg)$ by $p_i]$ that occurs when p_i invokes the operation (invocation or start event), and the event denoted $resp[X.\mathsf{op}(res)$ by $p_i]$ that occurs when the operation terminates (response, reply, or end event). We say that these events are (a) generated by the process p_i and (b) associated with the object X. Given an operation $X.\mathsf{op}(arg)(res)$, the event $resp[X.\mathsf{op}(res)$ by $p_i]$ is called the reply event matching the invocation event $inv[X.\mathsf{op}(arg)$ by $p_i]$.

Execution (or run) An execution of a multiprocess program induces a sequence of interactions between processes and concurrent objects. Every such interaction is represented by an *event*, i.e., the invocation of or the reply to an operation. A sequence of events is called a *history*, and this is precisely how executions are abstracted in the computation model. (The notion of history is detailed later in this chapter.)

Sequentiality of processes Each process is assumed to be *sequential*, which means that it executes one operation of an object at a time; that is, the algorithm of a sequential process stipulates that, after an operation is invoked on an object and until a matching reply is received, the process does not invoke any other operation. The fact that processes are each sequential does not preclude them from concurrently invoking operations on the same concurrent object. Sometimes, we focus on *sequential executions* (*sequential histories*), which precisely preclude such concurrency (only one process at a time invokes an operation on an object in a sequential execution). In this particular case, there is no overlapping of operation executions by different processes.

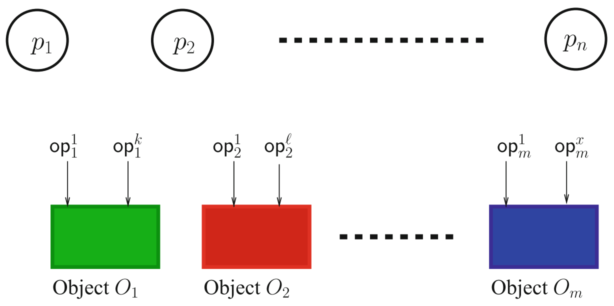


Fig. 4.3 Structural view of a system

4.2.2 Objects

An object has a name and a type. A type is defined by (1) the set of possible values for (the states of) objects of that type, (2) a finite set of operations through which the objects of that type can be manipulated, and (3) a specification describing, for each operation, the condition under which that operation can be invoked, and the effect produced after the operation was executed. Figure 4.3 presents a structural view of a set of *n* processes sharing *m* objects.

Sequential specification The object types we consider are defined by a *sequential specification*. (We talk interchangeably about the specification of the object or the specification of the type.) A sequential specification depicts the behavior of the object when accessed sequentially, i.e., in a sequential execution. This means that, despite concurrency, the implementation of any such object has to provide the illusion of sequential accesses. As already noticed, the aim is to facilitate the task of application programmers who have to reason only about sequential specifications.

It is common to define a sequential specification by associating two predicates with each operation. These predicates are called *pre-assertion* and *post-assertion*. Assuming the pre-assertion is satisfied before executing the operation, the post-assertion describes the new value of the object and the result of the operation returned to the calling process. We refine the notion of sequential specification in terms of histories later in this chapter.

Total versus partial operations An object operation is *total* if it is defined for every state of the object; otherwise it is *partial*. This means that, differently from a pre-assertion associated with a partial operation, the pre-assertion associated with a total operation is always satisfied.

Deterministic versus non-deterministic operations An object operation is *deterministic* if, given any state of the object that satisfies the pre-assertion of the operation, and given any valid input parameters of the operation, the output parameters and the final state of the object are uniquely defined. An object type that has only deterministic operations is said to be deterministic. (Objects of such a type are also said to be deterministic.) Otherwise, the object and its type are non-deterministic.

A few examples As we have seen in Chap. 1, an atomic read/write register is defined from a sequential operation and both its read and write operations are total.

Similarly, the unbounded FIFO queue defined in Sect. 4.1 can easily be defined from a sequential specification, and it has total operations (as the queue is unbounded, Q.enq(v) can always be executed, and as Q.deq() returns $\perp$ when the queue is empty, this operation also is total).

Let us consider a bounded queue Q such that Q.enq() is blocked when the queue is full, and Q.deq() is blocked when the queue is empty. Such a queue can easily be implemented by a monitor (see Chap. 3). It is easy to see that both Q.enq() and Q.deq() are partial. If the queue Q is unbounded, Q.enq() is total (because an invocation of Q.enq() cannot be blocked due to the fact that there is room to enqueue a new value) while Q.deq() is partial (because no value can be returned when the queue is empty).

To illustrate the idea of a non-deterministic object type, consider a bag. This type exports two operations: insert(e) (where e is an element from some value domain) and remove(). The first operation simply returns an indication, say *ok*, stipulating that the element was inserted into the bag. The second operation removes and returns *any* element from the bag. Hence, the current state of the bag does not uniquely determine which element will be removed, and this is the precise source of the non-determinism.

Finally, as we have seen in Chap. 1, a rendezvous object has no sequential specification.

4.2.3 Histories

The notion of an execution of a set of processes accessing concurrent objects is formally captured by the concept of a *history*.

Representing an execution as a history of events When considering events generated by sequential processes accessing concurrent objects with a sequential specification, it is always possible, without loss of generality, to arbitrarily order simultaneous events. This is because simultaneous (invocation and reply) events are independent (none of them can be the cause of the other). This observation makes it possible to consider a total order relation (denoted $<_H$ in the following) on the events of an execution abstracting the real-time order in which the events actually occur.

Hence, the interactions between a set of sequential processes and a set of shared objects are modeled by a sequence of invocation and reply events, called a *history* (sometimes also called a *trace*), and denoted $\widehat{H} = (H, <_H)$, where H is the set of events generated by the processes and $<_H$ a total order on these events. The objects and processes associated with events of $\widehat{H} = (H, <_H)$ are said to be involved in $\widehat{H}$. $\widehat{H}|p_i$ ($\widehat{H}$ at p_i) is called a *local* history; it denotes the sub-sequence of $\widehat{H}$ made up of all the events generated by the process p_i.

Complete versus partial histories An operation is said to be *complete* in a history if the history includes both the event corresponding to the invocation of the operation and its reply. Otherwise we say that the operation is *pending*. The fact that an operation is pending typically helps model the fact that a process is stopped by the operating system (paged out or swapped out) or simply *crashed*, say because the processor hosting that process incurs a physical fault.

A history without pending operations is said to be *complete*. A history with pending operations is said to be *partial*. Note that, being sequential, a process can have at most one pending operation in a given history.

Equivalent histories Two histories $\widehat{H}$ and $\widehat{H}'$ are said to be *equivalent* if they have the same local histories, i.e., for each p_i, $\widehat{H}|p_i = \widehat{H}'|p_i$. So, equivalent histories are built from the same set of events (remembering that an event includes the name of an object, the name of a process, the name of an operation, and input or output parameters).

Well-formed histories As histories are generated by sequential processes, we restrict our attention to histories $\widehat{H}$ such that, for each process p_i, $\widehat{H}|p_i$ is sequential. Such a local history starts with an invocation, followed by a matching reply, followed by another invocation, etc. The corresponding history $\widehat{H}$ is said to be *well-formed*.

Partial order on operations A history $\widehat{H}$ induces an irreflexive partial order on its operations as follows. Let $\mathsf{op} = X.\mathsf{op1}()$ by p_i and $\mathsf{op}' = Y.\mathsf{op2}()$ by p_j be two operations. Informally, operation op precedes operation op' if op terminates before op' starts, where "terminates" and "starts" refer to the time-line abstracted by the $<_H$ total order relation. More formally

$$\big(\mathsf{op} \rightarrow_H \mathsf{op}'\big) \overset{\text{def}}{=} \big(resp[\mathsf{op}] <_H inv[\mathsf{op}']\big).$$

Two operations op and op' are said to *overlap* (or *be concurrent*) in a history $\widehat{H}$ if neither $resp[\mathsf{op}] <_H inv[\mathsf{op}']$ nor $resp[\mathsf{op}'] <_H inv[\mathsf{op}]$. Notice that two overlapping operations are such that $\neg(\mathsf{op} \rightarrow_H \mathsf{op}')$ and $\neg(\mathsf{op}' \rightarrow_H \mathsf{op})$.

A sequential history has no overlapping operations; i.e., for any pair of operations op and op', we have $(\mathsf{op} \neq \mathsf{op}') \Rightarrow \big((\mathsf{op} \rightarrow_H \mathsf{op}') \vee ((\mathsf{op}' \rightarrow_H \mathsf{op})\big)$. $\rightarrow_H$ is consequently a total order if $\widehat{H}$ is a sequential history.

Illustrating histories Figure 4.4 depicts the (well-formed) history $\widehat{H}$ associated with the queue object execution described in Fig. 4.2. This history comprises ten events $e1 \ldots e10$ ($e4$, $e6$, $e7$, and $e9$ are explicitly detailed). As there is a single object, its name is omitted. Let us notice that the operation $\mathsf{enq}(c)$ by p_2 is concurrent with both $\mathsf{enq}(a)$ and $\mathsf{enq}(b)$ issued by p_1. Moreover, as the history $\widehat{H}$ has no pending operations, it is a complete history.

The sequence $e1 \ldots e9$ is a partial history where the dequeue operation issued by p_1 is pending. The sequence $e1 \ldots e6\, e7\, e8\, e10$ is another partial history in which the dequeue operation issued by p_2 is pending. Finally, the history $e1 \ldots e8$ has two pending operations.

4.2.4 Sequential History

Definition A history is *sequential* if its first event is an invocation, and then (1) each invocation event, except possibly the last, is immediately followed by the matching reply event, and (2) each reply event, except possibly the last, is immediately followed by an invocation event. The phrase "except possibly the last" associated with an invocation event is due to the fact that a history can be partial. A complete sequential history always ends with a reply event. A history that is not sequential is *concurrent*.

A sequential history models a sequential multiprocess computation (there are no overlapping operations in such a computation), while a concurrent history models a concurrent multiprocess computation (there are at least two overlapping operations in such a computation). Given that a sequential history $\widehat{S}$ has no overlapping operations, the associated partial order $\rightarrow_S$ defined on its operations is actually a total order. With a sequential history, one can thus reason about executions at the granularity of the operations invoked by the processes, instead of at the granularity of the underlying events.

Strictly speaking, the sequential specification of an object is a set of sequential histories involving solely that object. Basically, the sequential specification represents all possible sequential ways according to which the object can be accessed such that the pre-assertion and post-assertion of each of its operations are respected.

Example The history $\widehat{H} = e1\, e2 \cdots e10$ depicted in Fig. 4.4 is a complete concurrent history. On the other hand, the complete history

$$\widehat{H_1} = e1\ e3\ e4\ e6\ e2\ e5\ e7\ e9\ e8\ e10$$

is sequential: it has no overlapping operations. We can thus highlight its sequential nature by separating its operations using square brackets as follows:

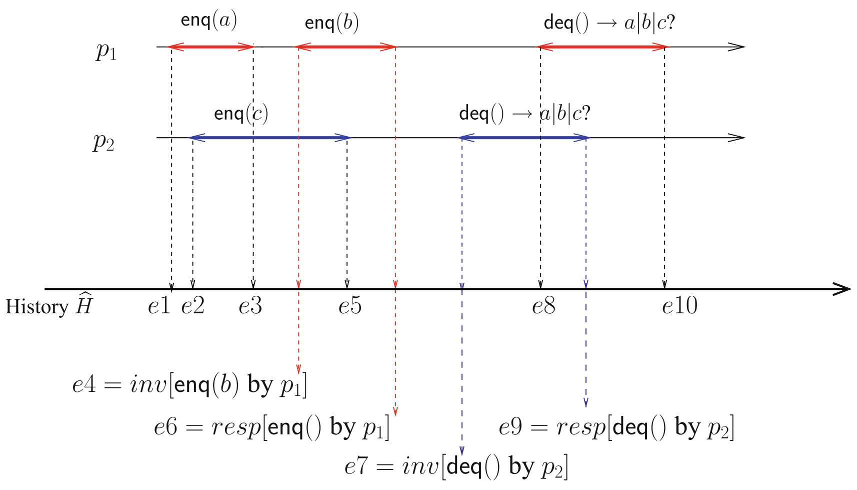


Fig. 4.4 Example of a history

$$\widehat{H_1} = [e1\, e3]\, [e4\, e6]\, [e2\, e5]\, [e7\, e9]\, [e8\, e10].$$

The histories

$$\widehat{H_2} = [e1\, e3]\, [e4\, e6]\, [e2\, e5]\, [e8\, e10]\, [e7\, e9],$$
$$\widehat{H_3} = [e1\, e3]\, [e4\, e6]\, [e8\, e10]\, [e2\, e5]\, [e7\, e9].$$

are also sequential. Let us also notice that $\widehat{H}$, $\widehat{H_1}$, $\widehat{H_2}$, $\widehat{H_3}$ are equivalent histories (they have the same local histories). Let $\widehat{H_4}$ be the history defined as

$$\widehat{H_4} = [e1\, e3]\, [e4\, e6]\, [e2\, e5]\, [e8\, e10]\, [e7\, e9].$$

$\widehat{H_4}$ is a partial sequential history. All these histories have the same local history for process p_1: $\widehat{H}|p_1 = \widehat{H_1}|p_1 = \widehat{H_2}|p_1 = \widehat{H_3}|p_1 = \widehat{H_4}|p_1 = [e1\, e3]\, [e4\, e6]\, [e8\, e10]$, and, as far p_2 is concerned, $\widehat{H_3}|p_2$ is a prefix of $\widehat{H}|p_2 = \widehat{H_1}|p_2 = \widehat{H_2}|p_2 = \widehat{H_3}|p_2 = [e2\, e5]\, [e7\, e9]$.

Hence, the notion of a history is an abstract way to depict the interactions between a set of processes and a set of concurrent objects. In short, a history is a *total order on the set of* (*invocation and reply*) *events* generated by the processes on the objects. As we are about to see, the notion of a history is central to defining the notion of *atomicity* through the very notion of *atomic history*.

4.3 Atomicity

The role of a correctness condition is to select, among all possible histories of a set of processes accessing shared objects, those considered to be *correct*. This section introduces the correctness condition called *atomicity* (also called *linearizability*). The aim of atomicity is to transform the difficult problem of reasoning about a concurrent execution into the simpler problem of reasoning about a sequential one.

Intuitively, atomicity states that a history is correct if its invocation and reply events could have been obtained, in the same order, by a single sequential process. In an atomic (or linearizable) history, each operation has to appear as if it was executed alone and instantaneously at some point between its invocation event and its reply event.

4.3.1 Legal History

As the concurrent objects that are considered are defined by sequential specifications, a definition of what is a "correct" history has to refer in one way or another to these specifications. The notion of *legal* history captures this idea.

Given a sequential history $\widehat{S}$, let $\widehat{S}|X$ ($\widehat{S}$ at X) denote the sub-sequence of $\widehat{S}$ made up of all the events involving object X. We say that a sequential history $\widehat{S}$ is *legal* if, for each object X, the sequence $\widehat{S}|X$ belongs to the sequential specification of X. In a sense, a history is legal if it could have been generated by processes accessing sequentially concurrent objects.

4.3.2 The Case of Complete Histories

This section first defines atomicity for complete histories $\widehat{H}$, i.e., histories without pending operations: each invocation event of $\widehat{H}$ has a matching reply event in $\widehat{H}$. The section that follows will extend this definition to partial histories.

Definition A complete history $\widehat{H}$ is *atomic* (or *linearizable*) if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_H \subseteq \rightarrow_S$.

The definition above states that, for a history $\widehat{H}$ to be linearizable, there must exist a permutation of $\widehat{H}$ (namely the witness history $\widehat{S}$) which satisfies the following requirements:

- First, $\widehat{S}$ has to be composed of the same set of events as $\widehat{H}$ and has to respect the local history of each process [item 1].
- Second, $\widehat{S}$ has to be sequential (interleave the process histories at the granularity of complete operations) and legal (respect the sequential specification of each object) [item 2]. Notice that, as $\widehat{S}$ is sequential, $\rightarrow_S$ is a total order.
- Finally, $\widehat{S}$ has also to respect the real-time occurrence order of the operations as defined by $\rightarrow_H$ [item 3].

$\widehat{S}$ represents a history that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of non-overlapping operations. Such a sequential history $\widehat{S}$ is called a *linearization* of $\widehat{H}$.

Proving that an algorithm implements an atomic object To this end, we need to prove that all histories generated by the algorithm are linearizable, i.e., identify a linearization of its operations that respects the "real-time" occurrence order of the operations and that is consistent with the sequential specification of the object.

It is important to notice that the notion of atomicity inherently includes a form of non-determinism. More precisely, given a history $\widehat{H}$, several linearizations of $\widehat{H}$ might exist.

Linearization: an example Let us consider the history $\widehat{H}$ described in Fig. 4.4 where the dequeue operation invoked by p_1 returns the value b while the dequeue operation invoked by p_2 returns the value a. This means that we have $e9 = resp[\mathsf{deq}(a) \text{ by } p_2]$ and $e10 = resp[\mathsf{deq}(b) \text{ by } p_1]$.

To show that this history is linearizable, we have to exhibit a witness history (linearization) satisfying the three requirements of atomicity. The reader can check that history

$$\widehat{H_1} = [e1\, e3]\ [e4\, e6]\ [e2\, e5]\ [e7\, e9]\ [e8\, e10]$$

defined in Sect. 4.2.4 is such a witness. At the granularity level defined by the operations, witness history $\widehat{H_1}$ can be represented as f

$$[\mathsf{enq}(a) \text{ by } p_1]\ [\mathsf{enq}(b) \text{ by } p_1]\ [\mathsf{enq}(c) \text{ by } p_2]\ [\mathsf{deq}(a) \text{ by } p_2]\ [\mathsf{deq}(b) \text{ by } p_1].$$

This formulation highlights the intuition that underlies the definition of the atomicity concept.

Linearization point The very existence of a linearization of an atomic history $\widehat{H}$ means that each operation of $\widehat{H}$ could have been executed at an indivisible instant between its invocation and reply time events (while providing the same result as $\widehat{H}$). It is thus possible to associate a *linearization point* with each operation of an atomic history. This is a point of the time line at which the corresponding operation could have been "instantaneously" executed according to the witness sequential and legal history.

To respect the real-time occurrence order, the linearization point associated with an operation has always to appear within the interval defined by the invocation event and the reply event associated with that operation.

Example Figure 4.5 depicts the linearization point of each operation. A triangle is associated with each operation, such that the vertex at the bottom of a triangle (bold dot) represents the associated linearization point. A triangle shows how atomicity allows shrinkage of an operation (the history of which takes some duration) into a single point on the time line.

In that sense, atomicity reduces the difficult problem of reasoning about a concurrent system to the simpler problem of reasoning about a sequential system where the operations issued by the processes are instantaneously executed.

As a second example, let us consider a variant of the history depicted in Fig. 4.5 where the reply events $e9$ and $e10$ are "exchanged", i.e., we have now that $e9 = resp[\mathsf{deq}(b) \text{ by } p_2]$ and $e10 = resp[\mathsf{deq}(a) \text{ by } p_1]$. It is easy to see that this history is linearizable: the sequential history $\widehat{H_2}$ described in Sect. 4.2.4 is a linearization of it.

Similarly, the history where $e9 = resp[\mathsf{deq}(c) \text{ by } p_2]$ and $e10 = resp[\mathsf{deq}(a) \text{ by } p_1]$ is also linearizable. It has the following sequential witness history:

$$[\mathsf{enq}(c) \text{ by } p_2]\ [\mathsf{enq}(a) \text{ by } p_1]\ [\mathsf{enq}(b) \text{ by } p_1]\ [\mathsf{deq}(c) \text{ by } p_2]\ [\mathsf{deq}(a) \text{ by } p_1].$$

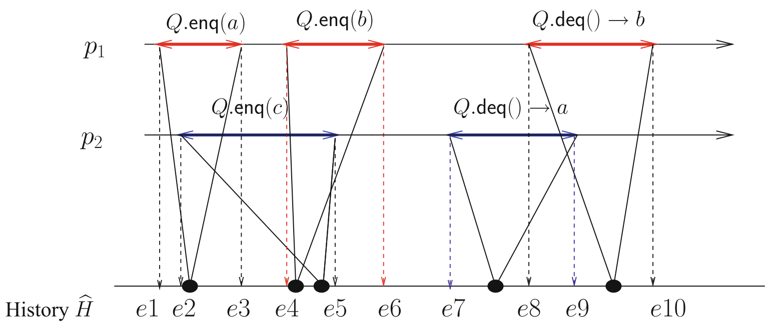


Fig. 4.5 Linearization points

Differently, the history in which the two dequeue operations would return the same value is not linearizable: it does not have a witness history which respects the sequential specification of the queue.

4.3.3 The Case of Partial Histories

This section extends the definition of atomicity to partial histories. As already indicated, these are histories with at least one process whose last operation is pending: the invocation event of this operation appears in the history while the corresponding reply event does not. The history $\widehat{H_4}$ described in Sect. 4.2.4 is a partial history. Extending atomicity to partial histories is important as it allows arbitrary delays experienced by processes, or even process crashes (when these delays become infinite), to be dealt with.

Definition A partial history $\widehat{H}$ is linearizable if $\widehat{H}$ can be modified in such a way that every invocation of a pending operation is either removed or completed with a reply event, and the resulting (complete) history $\widehat{H}'$ is linearizable.

Basically, the problem of determining whether a partial history $\widehat{H}$ is linearizable is reduced to the problem of determining whether a complete history $\widehat{H}'$, extracted from $\widehat{H}$, is linearizable. We obtain $\widehat{H}'$ by adding reply events to certain pending operations of $\widehat{H}$, as if these operations have indeed been completed, but also removing invocation events from some of the pending operations of $\widehat{H}$. We require, however, that all complete operations of $\widehat{H}$ be preserved in $\widehat{H}'$. It is important to notice that, given a history $\widehat{H}$, we can extract several histories $\widehat{H}'$ that satisfy the required conditions.

Example Let us consider Fig. 4.6, which depicts two processes accessing a register. Process p_1 first writes the value 0. The same process later issues a write for the value 1, but p_1 crashes during this second write (this is indicated by a cross on its time line). Process p_2 executes two consecutive read operations. The first read operation lies between the two write operations of p_1 and returns the value 0. A different value would clearly violate atomicity. The situation is less obvious with the second value,

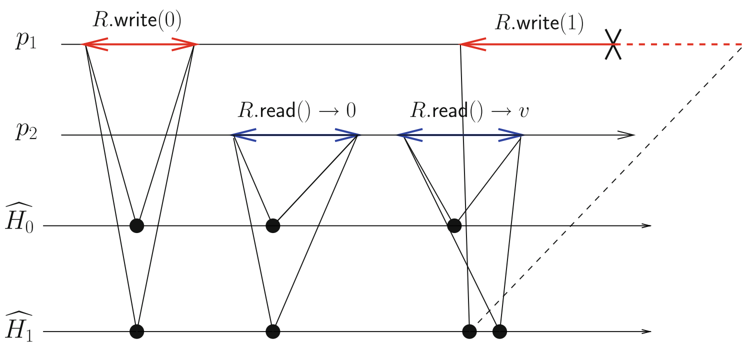


Fig. 4.6 Two ways of completing a history

and it is not entirely clear what value v has to be returned by the second read operation in order for the history to be linearizable.

As explained below, both values 0 and 1 can be returned by that read operation while preserving atomicity. The second write operation is pending in the partial history $\widehat{H}$ modeling this execution. This history $\widehat{H}$ is made up of seven events (the names of the object and the processes are omitted as there is no ambiguity), namely

$$inv[\mathsf{write}(0)]\ resp[\mathsf{write}(0)]\ inv[\mathsf{read}(0)]\ resp[\mathsf{read}(0)]$$
$$inv[\mathsf{read}(v)]\ inv[\mathsf{write}(1)]\ resp[\mathsf{read}(v)].$$

We explain now why both 0 and 1 can be returned by the second read:

- Let us first assume that the returned value v is 0.
  We can associate with history $\widehat{H}$ a legal sequential witness history $\widehat{H_0}$ which includes only complete operations and respects the partial order defined by $\widehat{H}$ on these operations (see Fig. 4.6). To obtain $\widehat{H_0}$, we construct history $\widehat{H}'$ by removing the event $inv[\mathsf{write}(1)]$ from $\widehat{H}$: we obtain a complete history, i.e., a history without pending operations.

  History $\widehat{H}$ with $v = 0$ is consequently linearizable. The associated witness history $\widehat{H_0}$ models the situation where p_1 is considered as having crashed before invoking the second write operation: everything appears as if this write had never been issued.

- Assume that the returned value v is 1.
  Similarly to the previous case, we can associate with history $\widehat{H}$ a witness legal sequential history $\widehat{H_1}$ that respects the partial order on the operations. We actually derive $\widehat{H_1}$ by first constructing $\widehat{H}'$, which we obtain by adding to $\widehat{H}$ the reply event $res[\mathsf{write}(1)]$. (In Fig. 4.6, the part added to $\widehat{H}$ in order to obtain $\widehat{H}'$—from which $\widehat{H_1}$ is constructed—is indicated by dotted lines.)

  The history where $v = 1$ is consequently linearizable. The associated witness history $\widehat{H_1}$ represents the situation where the second write is taken into account despite the crash of the process that issued that write operation.

4.4 Object Composability and Guaranteed Termination Property

This section presents two fundamental properties of atomicity that make it particularly attractive. The first property states that atomic objects can be composed for free, while the second property states that, as object operations are total, no operation invocation can be prevented from terminating.

4.4.1 Atomic Objects Compose for Free

The notion of a local property Let P be any property defined on a set of objects. The property P is said to be *local* if the set of objects as a whole satisfies P whenever each object taken alone satisfies P.

Locality is an important concept that promotes modularity. Consider some local property P. To prove that an entire set of objects satisfy P, we only have to ensure that each object, independently from the others, satisfies P. As a consequence, the property P can be implemented for each object independently of the way it implemented for the other objects. At one extreme, it is even possible to design an implementation where each object has its own algorithm implementing P. At another extreme, all the objects (whatever their type) might use the same algorithm to implement P (each object using its own instance of the algorithm).

Atomicity is a local property We prove in the following that atomicity is a local property. Intuitively, the fact that atomicity is local comes from the fact that it involves the real-time occurrence order on non-concurrent operations whatever the objects and the processes concerned by these operations. We will rely on this aspect in the proof of the following theorem.

Theorem 14 *A history $\widehat{H}$ is atomic (linearizable) if and only if, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is atomic (linearizable).*

Proof The "$\Rightarrow$" direction (only if) is an immediate consequence of the definition of atomicity: if $\widehat{H}$ is linearizable then, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is linearizable. So, the rest of the proof is restricted to the "$\Leftarrow$" direction. We also restrict the rest of the proof to the case where $\widehat{H}$ is complete, i.e., $\widehat{H}$ has no pending operation. This is without loss of generality, given that the definition of atomicity for a partial history is derived from the definition of atomicity for a complete history.

Given an object X, let $\widehat{S_X}$ be a linearization of $\widehat{H}|X$. It follows from the definition of atomicity that $\widehat{S_X}$ defines a total order on the operations involving X. Let $\rightarrow_X$ denote this total order. We construct an order relation $\rightarrow$ defined on the whole set of operations of $\widehat{H}$ as follows:

1. For each object X: $\rightarrow_X \subseteq \rightarrow$,
2. $\rightarrow_H \subseteq \rightarrow$.

Basically, "$\rightarrow$" totally orders all operations on the same object X, according to $\rightarrow_X$ (item 1), while preserving $\rightarrow_H$, i.e., the real-time occurrence order on the operations (item 2). □

Claim. "$\rightarrow$ is acyclic". This claim means that $\rightarrow$ defines a partial order on the set of all the operations of $\widehat{H}$.

Assuming this claim (see its proof below), it is thus possible to construct a sequential history $\widehat{S}$ including all events of $\widehat{H}$ and respecting $\rightarrow$. We trivially have $\rightarrow \subseteq \rightarrow_S$, where $\rightarrow_S$ is the total order on the operations defined from $\widehat{S}$. We have the three following conditions: (1) $\widehat{H}$ and $\widehat{S}$ are equivalent (they contain the same events and contain the same local histories), (2) $\widehat{S}$ is sequential (by construction) and legal (due to item 1 above), and (3) $\rightarrow_H \subseteq \rightarrow_S$ (due to item 2 above and $\rightarrow \subseteq \rightarrow_S$). It follows that $\widehat{H}$ is linearizable.

Proof of the claim. We show (by contradiction) that $\rightarrow$ is acyclic. Assume first that $\rightarrow$ induces a cycle involving the operations on a single object X. Indeed, as $\rightarrow_X$ is a total order, in particular transitive, there must be two operations op_i and op_j on X such that $\mathsf{op}_i \rightarrow_X \mathsf{op}_j$ and $\mathsf{op}_j \rightarrow_H \mathsf{op}_i$. But $\mathsf{op}_i \rightarrow_X \mathsf{op}_j \Rightarrow inv[\mathsf{op}_i] <_H resp[\mathsf{op}_j]$ because X is linearizable. As $<_H$ is a total order on the whole set of events, the fact that $\mathsf{op}_j \rightarrow_H \mathsf{op}_i \Rightarrow resp[\mathsf{op}_j] <_H inv[\mathsf{op}_i]$ establishes the contradiction.

It follows that any cycle must involve at least two objects. To obtain a contradiction we show that, in that case, a cycle in $\rightarrow$ implies a cycle in $\rightarrow_H$ (which is acyclic). Let us examine the way the cycle could be obtained. If two consecutive edges of the cycle are due to just some $\rightarrow_X$ or just $\rightarrow_H$, then the cycle can be shortened, as any of these relations is transitive. Moreover, $\mathsf{op}_i \rightarrow_X \mathsf{op}_j \rightarrow_Y \mathsf{op}_k$ is not possible for $X \neq Y$, as each operation is on only one object ($\mathsf{op}_i \rightarrow_X \mathsf{op}_j \rightarrow_Y \mathsf{op}_k$ would imply that op_j is on both X and Y). So let us consider any sequence of edges of the cycle such that: $\mathsf{op}_1 \rightarrow_H \mathsf{op}_2 \rightarrow_X \mathsf{op}_3 \rightarrow_H \mathsf{op}_4$. We have:

– $\mathsf{op}_1 \rightarrow_H \mathsf{op}_2 \Rightarrow resp[\mathsf{op}_1] <_H inv[\mathsf{op}_2]$ (definition of $\mathsf{op}_1 \rightarrow_H$),
– $\mathsf{op}_2 \rightarrow_X \mathsf{op}_3 \Rightarrow inv[\mathsf{op}_2] <_H resp[\mathsf{op}_3]$ (as X is linearizable),
– $\mathsf{op}_3 \rightarrow_H \mathsf{op}_4 \Rightarrow resp[\mathsf{op}_3] <_H inv[\mathsf{op}_4]$ (definition of $\mathsf{op}_1 \rightarrow_H$).

Combining these statements, we obtain $resp[\mathsf{op}_1] <_H inv[\mathsf{op}_4]$, from which we can conclude that $\mathsf{op}_1 \rightarrow_H \mathsf{op}_4$. It follows that any cycle in $\rightarrow$ can be reduced to a cycle in $\rightarrow_H$, which is a contradiction as $\rightarrow_H$ is an irreflexive partial order. *End of the proof of the claim.* □

The benefit of locality Considering an execution of a set of processes that access concurrently a set of objects, atomicity allows the programmer to reason as if all the operations issued by the processes on the objects were executed one after the other. The previous theorem is fundamental. It states that, to reason about sequential processes that access concurrent atomic objects, one can reason on each object independently, without losing the atomicity property of the whole computation.

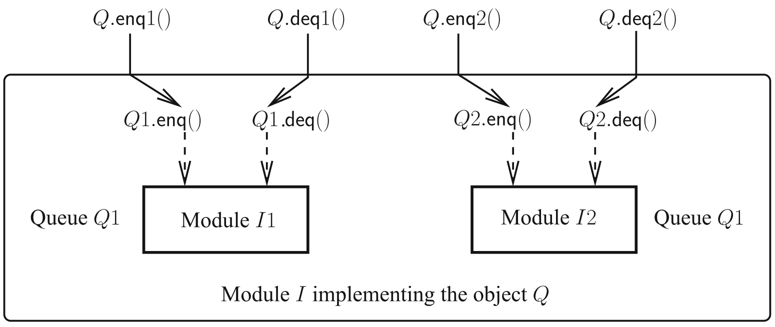


Fig. 4.7 Atomicity allows objects to compose for free

An example Locality means that atomic objects compose for free. As an example, let us consider two atomic queue objects $Q1$ and $Q2$ each with its own implementation $I1$ and $I2$, respectively (hence, the implementations can use different algorithms).

Let us define the object Q that is a composition of $Q1$ and $Q2$ defined as follows (Fig. 4.7). Q provides processes with the four following operations Q.enq1(), Q.deq1(), Q.enq2(), and Q.deq2() whose effect is the same as $Q1$.enq(), $Q1$.deq(), $Q2$.enq() and $Q2$.deq(), respectively.

Thanks to locality, an implementation of Q consists simply in piecing together $I1$ and $I2$ *without any modification* to their code. As we will see in Sect.4.5, this object composition property is no longer true for other consistency conditions.

4.4.2 Guaranteed Termination

Due to the fact that operations are total, atomicity (linearizability) per se does not require a pending invocation of an operation to wait for another operation to complete. This means that, if a given implementation I of an atomic object entails blocking of a total operation, this is not due to the atomicity concept but only to I. Blocking is an artifact of particular implementations of atomicity but not an inherent feature of atomicity.

This property of the atomicity consistency condition is captured by the following theorem, which states that any (atomic) history with a pending operation invocation can be extended with a reply to that operation.

Theorem 15 *Let* $inv[\mathsf{op}(arg)]$ *be the invocation event of a total operation that is pending in a linearizable history* $\widehat{H}$. *There exists a matching reply event* $res[\mathsf{op}(res)]$ *such that the history* $\widehat{H}' = \widehat{H}.resp[\mathsf{op}(res)]$ *is linearizable.*

Proof Let $\widehat{S}$ be a linearization of the partial history $\widehat{H}$. By definition of a linearization, $\widehat{S}$ has a matching reply to every invocation. Assume first that $\widehat{S}$ includes a reply event $resp[\mathsf{op}(res)]$ matching the invocation event $inv[\mathsf{op}(arg)]$. In this case, the theorem trivially follows, as then $\widehat{S}$ is also a linearization of $\widehat{H}'$.

If $\widehat{S}$ does not include a matching reply event, then $\widehat{S}$ does not include $inv[\mathsf{op}(arg)]$. Because the operation op() is total, there is a reply event $resp[\mathsf{op}(res)]$ matching the invocation event $inv[\mathsf{op}(arg)]$ in every state of the shared object. Let $\widehat{S'}$ be the sequential history $\widehat{S}$ with the invocation event $inv[op(arg)]$ and a matching reply event $resp[\mathsf{op}(res)]$ added in that order at the end of $\widehat{S}$. $\widehat{S'}$ is trivially legal. It follows that $\widehat{S'}$ is a linearization of $\widehat{H'}$. □

4.5 Alternatives to Atomicity

This section discusses two alternatives to atomicity, namely *sequential consistency* and *serializability*.

4.5.1 Sequential Consistency

Overview Both atomicity and sequential consistency guarantee that operations appear to execute instantaneously at some point on the time line. The difference is that atomicity requires that, for each operation, this instant lies between the occurrence times of the invocation and reply events associated with the operation, which is not the case for sequential consistency.

More precisely, the definition of atomicity requires that the witness sequential history that is equivalent to a history $\widehat{H}$ respects the partial order relation on operations of $\widehat{H}$ (also called the real-time order). This is irrespective of the process and the object involved in the operations. Sequential consistency is a weaker property in the sense that it requires only for the witness history to preserve the order on the operations invoked by the same process.

Let $<_i$ denote the total order on the events generated by process p_i (this order corresponds to the projection of $\widehat{H}$ on p_i denoted $\widehat{H}|p_i$ in Sect. 4.2.3). Moreover, let $<_{proc} = \cup_{1\leq i\leq n} <_i$. This is the union of the n total orders associated with the processes, which is called the *process-order* relation.

To illustrate this relation, consider Fig. 4.4, where $<_{proc}$ is the union of the local history of p_1, $<_1$, and the local history of p_2, $<_2$, namely

$$<_1 = [e1\, e3]\, [e4\, e6]\, [e8\, e10] \quad \text{and} \quad <_2 = [e2\, e5]\, [e7\, e9].$$

It is easy to see that $\widehat{H}$ can be partitioned into two partial order relations, namely $<_{proc}$ and $<_{object}$ ($<_{object}$ is $\widehat{H}$ from which the edges due to $<_{proc}$ have been suppressed).

As pointed out above, in contrast to atomicity that establishes the correctness of a history using the constraints imposed by both $<_{proc}$ and $<_{object}$, sequential consistency establishes correctness based only on the process-order relation (i.e., $<_{proc}$).

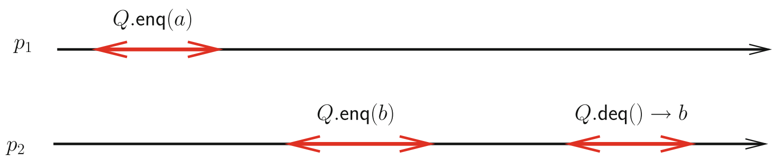


Fig. 4.8 A sequentially consistent history

Definition The definition of the sequential consistency correctness condition reuses the notions of history, sequential history, complete history, as in Sect. 4.2. To simplify the presentation and without loss of generality, we only consider complete histories (with no pending operations). Considering $<_{proc}$ to be the process order relation on the events, we also define $\rightarrow_{proc}$ as the partial order on the operations induced from $<_{proc}$.

A history $\widehat{H}$ is *sequentially consistent* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_{proc} \subseteq \rightarrow_S$ ($\widehat{S}$ has to respect process-order).

To illustrate sequential consistency, consider Fig. 4.8. There are two processes p_1 and p_2 that share a queue Q. At the operation level, the local history of p_1 comprises a single operation, Q.enq(a), while the local history of p_2 comprises two operations, first Q.enq(b) and then Q.deq$()$, which returns b. The reader can easily verify that this history is not atomic. This is because, for atomicity, all the operations must totally ordered according to real time. Consequently the Q.deq$()$ operation issued by p_2 should return the value a whose enqueuing was terminated before the enqueuing of a had started.

However, the history is sequentially consistent: the sequential history (described at the operation level)

$$\widehat{S} = [Q.\mathsf{enq}(b) \text{ by } p_2]\ [Q.\mathsf{enq}(a) \text{ by } p_1]\ [Q.\mathsf{deq}(b) \text{ by } p_2]$$

is legal and respects the process-order relation.

Atomicity versus sequential consistency It is easy to see from the previous definition that any linearizable history is also sequentially consistent: this is because $\rightarrow_{proc} \subseteq \rightarrow_H$. As shown by the example of Fig. 4.8 however, the contrary is not true. It is then natural to ask whether sequential consistency would not be sufficient to judge correctness.

A drawback of sequential consistency is that it is not a local property. (This is the price that sequential consistency has to pay to allow for more correct executions than atomicity.) To illustrate this, consider the counter-example described in Fig. 4.9. History $\widehat{H}$ involves two processes accessing two concurrent queues Q and Q'. It is

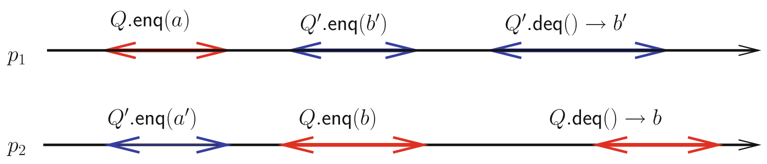


Fig. 4.9 Sequential consistency is not a local property

easy to see that, when we consider each object in isolation, we obtain the histories $\widehat{H}|Q$ and $\widehat{H}|Q'$ that are sequentially consistent. Unfortunately, there is no way to witness a legal total order $\widehat{S}$ that involves the six operations: if p_1 dequeues b' from Q', Q'.enq(a') has to be ordered after Q'.enq(b') in a witness sequential history. But this means that (to respect process-order) Q.enq(a) by p_1 is necessarily ordered before Q.enq(b) by p_2. Consequently Q.deq() by p_2 should return a for $\widehat{S}$ to be legal. A similar reasoning can be done starting from the operation Q.deq(b) by p_2. It follows that there can be no legal witness total order. Hence, despite the fact that $\widehat{H}|Q$ and $\widehat{H}|Q'$ are sequentially consistent, the whole history $\widehat{H}$ is not.

4.5.2 Serializability

Overview It is sometimes important to ensure that *groups* of operations appear to execute as if they have been executed without interference with any other group of operations. The concept of *transaction* is then the appropriate abstraction that allows the grouping of operations. This abstraction is mainly encountered in database systems.

A transaction is a sequence of operations that might complete successfully (commit) or abort. In short, the execution of a set of concurrent transactions is correct if committed transactions appear to execute at some indivisible point in time and aborted transactions do not appear to have been executed at all. This correctness criteria is called *serializability* (sometimes it is also called atomicity). The motivation (again) is to reduce the difficult problem of reasoning about concurrent transactions into the easier problem of reasoning about transactions that are executed one after the other. For instance, if some invariant predicate on the set of shared objects is preserved by every individual committed transaction, then it will be preserved by a serializable execution of transactions.

Definition To define serializability, the notion of history needs to be revisited. Events are now associated with objects and transactions. In short, processes are replaced by transactions. For each transaction, in addition to the invocation and reply events, two new events come into the picture: *commit* and *abort* events. These are associated with transactions. At most one such event is associated with every transaction in a history. A transaction without such an event is called pending; otherwise the transaction is said to be *complete* (committed or aborted). Adding a *commit* (or *abort*) event after all other events of a pending transaction is called committing (or aborting) the transaction. A sequential history is a sequence of committed transactions.

Let $\rightarrow_{trans}$ denote the total order of events of the committed transactions. This is analogous to the process-order relation defined above. We say that a history is *complete* if all its transactions are complete.

Let $\widehat{H}$ be a complete history. $\widehat{H}$ is *serializable* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{S}$ is made up of all events of committed transactions of $\widehat{H}$,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_{trans} \subseteq \rightarrow_{S}$ ($\widehat{S}$ has to respect transaction order).

Let $\widehat{H}$ be a history that is not complete. $\widehat{H}$ is *serializable* if we can derive from $\widehat{H}$ a complete history $\widehat{H}'$ (by completing or removing pending transactions from $\widehat{H}$) such that: (1) $\widehat{H}'$ is complete, (2) $\widehat{H}'$ includes the complete transactions of $\widehat{H}$, and (3) $\widehat{H}'$ is serializable.

Atomicity versus serializability As for atomicity, serializability is defined according to the equivalence to a witness sequential history, but differently from atomicity, no real-time ordering is required. In this sense, serializability can be viewed as an extension of sequential consistency to transactions where a transaction is made up of several invocations of object operations. Unlike atomicity, serializability is not a local property (replacing processes with transactions in Fig. 4.9 gives a counter-example).

4.6 Summary

This chapter has introduced the basic elements that are needed to reason about executions of a multiprocess program whose processes cooperate through concurrent objects (defined by a sequential specification on total operations). More specifically, this chapter has presented the basic notions from which the atomicity concept has then been defined.

The fundamental modeling element is that of a history: a sequence of events depicting the interaction between processes and objects. An event represents the invocation of an object or the return of a reply. A history is atomic if, despite concurrency, it appears as if processes access the objects by invoking operations one after the other. In this sense, the correctness of a concurrent computation is judged with respect to a sequential behavior, itself determined by the sequential specification of the objects. Hence, atomicity is what allows us to reason sequentially despite concurrency.

4.7 Bibliographic Notes

- The notion of atomic read/write objects (registers), as studied here, was investigated and formalized by L. Lamport [189] and J. Misra [206].
- The generalization of the atomicity consistency condition to objects of any sequential type was developed by M. Herlihy and J. Wing under the name linearizability [148].
- The notion of sequential consistency was introduced by L. Lamport [187].

  The relation between atomicity and sequential consistency was investigated in [40] and [232], where it was shown that, from a protocol design point of view, sequential consistency can be seen as lazy linearizability. Examples of protocols implementing sequential consistency can be found in [3, 40, 233].
- The concept of transactions is part of almost every textbook on database systems. Books entirely devoted to transactions include [50, 97, 119]. The theory of serializability is the main topic of [97, 222].

Part III
Mutex-Free Synchronization

While Part I was devoted to lock-based synchronization, this part of the book is on the design of concurrent objects whose implementation does not rely on mutual exclusion. It is made up of five chapters:

- The first chapter introduces the notion of a mutex-free implementation (i.e., implementations which are not allowed to rely on locks) and the associated liveness properties, namely obstruction-freedom, non-blocking, and wait-freedom.
- The second chapter introduces the notion of a hybrid implementation, namely an implementation which is partly lock-based and partly mutex-free.
- The next three chapters are on the power of atomic read/write registers when one has to design wait-free object implementations. These chapters show that non-trivial objects can be built in such a particularly poor context. To that end, they present wait-free implementations of the following concurrent objects: weak counters, store-collect objects, snapshot objects, and renaming objects.

Remark on terminology As we are about to see, the term *mutex-freedom* is used to indicate that the use of critical sections (locks) is prohibited. The term *lock-freedom* could have been used instead of *mutex-freedom*. This has not been done for the following reason: the term lock-freedom is already used in a lot of papers on synchronization with different meanings. In order not to overload it and to prevent confusion, the term *mutex-freedom* is used in this book.

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 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).

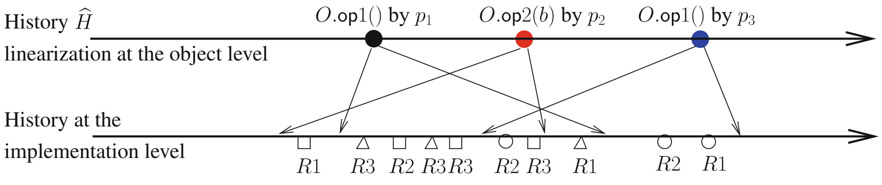


Fig. 5.1 Interleaving at the implementation level

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:

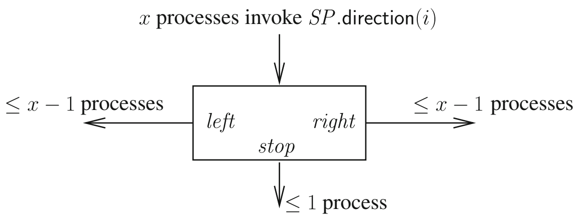


Fig. 5.2 Splitter object

- 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)  $LAST \leftarrow i$;
(2)  if $(DOOR = closed)$
(3)     then return($right$)
(4)     else  $DOOR \leftarrow closed$;
(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*

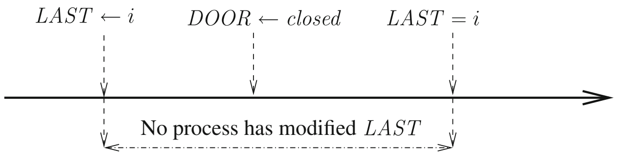


Fig. 5.4 On the modification of *LAST*

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 $\text{gt}_1()$ and $\text{gt}_2()$ be two distinct invocations of get_timestamp(). If $\text{gt}_1()$ returns before $\text{gt}_2()$ starts, the timestamp returned by $\text{gt}_2()$ is greater than the one returned by $\text{gt}_1()$.
- 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)  $k \leftarrow NEXT$;
(2)  repeat forever
(3)     $LAST[k] \leftarrow i$;
(4)     if $(\neg COMP[k])$
(5)        then $COMP[k] \leftarrow true$;
(6)             if $(LAST[k] = i)$ then $NEXT \leftarrow NEXT + 1$; return($k$) end if
(7)     end if;
(8)  $k \leftarrow k + 1$
(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 ← 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 ← 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.

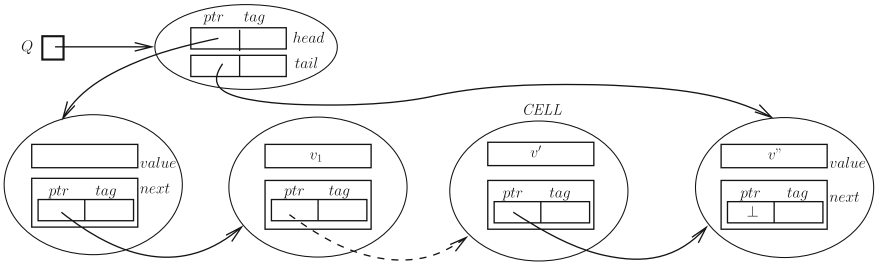


Fig. 5.7 The list implementing the queue

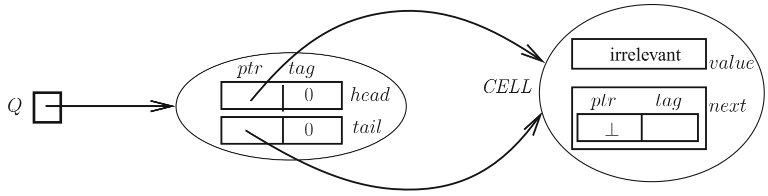


Fig. 5.8 Initial state of the list

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)  $pt\_cell \leftarrow\uparrow$ new_cell(); $(pt\_cell \downarrow).value \leftarrow v$; $(pt\_cell \downarrow).next.ptr \leftarrow \bot$;
(2)  repeat forever
(3)     $\ell tail \leftarrow (Q \downarrow).tail$;
(4)     $\ell next \leftarrow (\ell tail.ptr \downarrow).next$;
(5)     if ($\ell tail = (Q \downarrow).tail$) then
(6)       if ($\ell next.ptr = \bot$)
(7)         then if $((\ell tail.ptr \downarrow).next)$.compare&swap($\ell next, \langle pt\_cell, \ell next.tag + 1\rangle$)
(8)                then($(Q \downarrow).tail$).compare&swap($\ell tail, \langle pt\_cell, \ell tail.tag + 1\rangle$); return($ok$)
(9)              end if
(10)        else  ($(Q \downarrow).tail$).compare&swap($\ell tail, \langle \ell next.ptr, \ell tail.tag + 1\rangle$)
(11)      end if
(12)    end if
(13) end repeat
end operation.

operation deq() is
(14) repeat forever
(15)    $\ell head \leftarrow (Q \downarrow).head$;
(16)    $\ell tail \leftarrow (Q \downarrow).tail$;
(17)    $\ell next \leftarrow (\ell head.ptr \downarrow).next$;
(18)    if ($\ell head = (Q \downarrow).head$) then
(19)      if ($\ell head.ptr = \ell tail.ptr$)
(20)        then if ($\ell next.ptr = \bot$) then return($empty$) end if;
(21)             ($(Q \downarrow).tail$).compare&swap($\ell tail, \langle \ell next.ptr, \ell tail.tag + 1\rangle$)
(22)        else $result \leftarrow (\ell next.ptr \downarrow).value$;
(23)             if ($(Q \downarrow).head$).compare&swap($\ell head, \langle \ell next.ptr, \ell head.tag + 1\rangle$)
(24)               then free($\ell head.ptr$); 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, $STACK[x]$ $(1 \le x \le k)$ 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 $newtop = \langle index + 1, v, sn\_of\_next + 1\rangle$, 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 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 $\langle index, value, seqnb\rangle$ it has read from *TOP* at line 2 or 11 is still in *TOP* when it executes *TOP*.compare&swap ($\langle index, value, seqnb\rangle, newtop$) 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($\langle index, value, seqnb\rangle, newtop$) 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)      $(index, value, seqnb) \leftarrow TOP$;
(3)      help($index, value, seqnb$);
(4)      if $(index = k)$ then return($full$) end if;
(5)      $sn\_of\_next \leftarrow STACK[index + 1].sn$;
(6)      $newtop \leftarrow \langle index + 1, v, sn\_of\_next + 1\rangle$;
(7)      if $TOP$.compare&swap$\big(\langle index, value, seqnb\rangle, newtop\big)$ then return($done$) end if
(8)  end repeat
end operation.

operation pop() is
(9)  repeat forever
(10)     $(index, value, seqnb) \leftarrow TOP$;
(11)     help($index, value, seqnb$);
(12)     if $(index = 0)$ then return($empty$) end if;
(13)     $belowtop \leftarrow STACK[index - 1]$;
(14)     $newtop \leftarrow \langle index - 1, belowtop.val, belowtop.sn + 1\rangle$;
(15)     if $TOP$.compare&swap$\big(\langle index, value, seqnb\rangle, newtop\big)$ then return($value$) end if
(16) end repeat
end operation.

procedure help($index, value, seqnb$):
(17) $stacktop \leftarrow STACK[index].val$;
(18) $STACK[index]$.compare&swap$\big(\langle stacktop, seqnb - 1\rangle, \langle value, seqnb\rangle\big)$
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)  $in \leftarrow NEXT$.fetch&add() $- 1$;
(2)  $REG[in] \leftarrow v$;
(3)  return()
end operation.

operation $Q$.pop() is
(4)  $last \leftarrow NEXT - 1$;
(5)  for $x$ from $last$ to 0 do
(6)      $aux \leftarrow REG[x]$.swap($\perp$);
(7)      if ($aux \neq \perp$) 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 ⊥. In this case, from an external observer point of view, everything appears as if the process crashed before invoking *STACK*.push(v).

From an internal point of view, the crash of p_i just before executing *REG*[*in*] ← v entails an increase of *NEXT*. But as the corresponding entry of the array *REG* will remain forever equal to ⊥, 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*.pop(). If p_i crashes after it has executed the statement *aux* ← *REG*[x].swap(⊥) (line 6), which has returned it a value, everything appears to an external observer as if p_i crashed after the invocation of *STACK*.pop(). In the other case, everything appears to an external observer as if p_i crashed before the invocation of *STACK*.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*.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*.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 push(c) obtained the index value x (at line 1) before the invocations of push(d), etc., 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] ← 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 push(c) is not the time at which it executes *REG*[x] ← c but a time instant just before the linearization points associated with push(d).

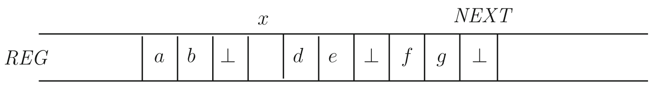


Fig. 5.12 On the linearization points of the wait-free stack

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

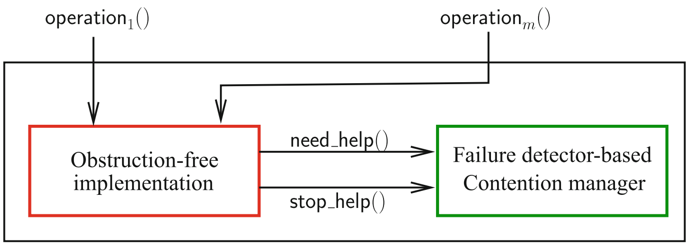

operation$_1$()
operation$_m$()
Obstruction-free implementation
need_help()
stop_help()
Failure detector-based Contention manager

Fig. 5.13 Boosting obstruction-freedom

```
operation get_timestamp($i$) is
(1)   $k \leftarrow NEXT$;
(2)   repeat forever
(3)      $LAST[k] \leftarrow i$;
(4)      if $(\neg COMP[k])$
(5)        then $COMP[k] \leftarrow true$;
(6)             if $(LAST[k] = i)$ then $NEXT \leftarrow NEXT + 1$; $\underline{CM.\mathsf{stop\_help}(i)}$; return($k$) end if
(7)      end if;
(8)   $k \leftarrow k + 1$; $\underline{CM.\mathsf{need\_help}(i)}$
(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 $[false, \ldots, false]$.

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
    $NEED\_HELP[i] \leftarrow true$;
    repeat $x \leftarrow \{j \mid NEED\_HELP[j]\}$ until $(ev\_leader(x) = i)$ end repeat;
    return()
end operation.

operation $CM$.stop_help($i$) is $NEED\_HELP[i] \leftarrow false$; 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. $\square$

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 $TS[i] \leftarrow$ weak_ts() end if;
(2)  repeat $competing \leftarrow \{j \mid TS[j] \neq 0) \wedge j \notin suspected\}$;
(3)           let $\langle ts, j\rangle$ be the smallest pair $\in \{\langle TS[x], x\rangle \mid x \in competing\}$
(4)  until ($j = i$) end repeat
end operation.

operation stop_help($i$) is $TS[i] \leftarrow 0$; 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.)

Chapter 6
Hybrid Concurrent Objects

This chapter focuses on *hybrid implementations* of concurrent objects. Roughly speaking, "hybrid" means that lock-based code and mutex-free code can be merged in the same implementation. After defining the notion of a hybrid implementation, this chapter presents hybrid implementations of concurrent objects, where each implementation has its own features. The chapter presents also the notion of an abortable object and shows how a starvation-free implementation of a concurrent object can be systematically obtained from an abortable non-blocking version of the same object.

Keywords Abortable object · Binary consensus · Concurrent set object · Contention sensitive implementation · Double-ended queue · Hybrid (static versus dynamic) implementation · LL/SC primitive operations · Linearization point · Non-blocking to starvation-freedom.

6.1 The Notion of a Hybrid Implementation

A hybrid implementation of an object can be seen as an "impure" mutex-free implementation in the sense that it is allowed to use locks for a subset of the operations or in specific concurrency patterns. Of course, a hybrid implementation is no longer fully mutex-free. Two kinds of hybrid implementations can be distinguished: *static hybrid* implementations and *dynamic hybrid* implementations (also called sensitive implementations).

As each object has its own internal representation R, both mutex-free operations and lock-based operations on that object can concurrently access R. It follows that the main difficulty in the design of hybrid implementations lies in the correct cooperation between mutex-free code and lock-based code.

6.1.1 Lock-Based Versus Mutex-Free Operation: Static Hybrid Implementation

This family of hybrid implementations considers two types of implementations for the operations associated with an object, namely operations whose implementation uses locks and operations whose implementation is wait-free. In the extreme case where all operations are lock-based, so is the implementation. Similarly, if all operations are mutex-free, so is the implementation. This type of hybrid implementation is called *static*, as the operations which are allowed to use locks and those which are not are statically determined.

Designing such hybrid implementations for concurrent objects is interesting when (a) there are no failures (this constraint is due to the use of locks) and (b) some operations are invoked much more than the others (the operations which are invoked very often being designed with efficient wait-free algorithms). A static hybrid implementation of a concurrent set object is presented in Sect. 6.2. In this set object, the operation that checks if an element belongs to the set is wait-free while the operation that adds an element to the set and the operation that removes an element from the set (which are assumed to be infrequent) are lock-based.

6.1.2 Contention Sensitive (or Dynamic Hybrid) Implementation

Contention sensitiveness is another notion of a hybrid implementation with captures the notion of a *dynamic* hybrid implementation. It states that, while any algorithm that implements an operation can potentially use locks, the overhead introduced by locks has to be eliminated in "favorable circumstances". Examples of "favorable circumstances" are the parts of an execution without concurrency, or the parts of an execution in which only processes with non-interfering operations access the object.

Hence, contention sensitiveness of an implementation depends on what is defined as "favorable circumstances" which in turn depends on the object that we want to implement. This point will be developed in Sects. 6.3 and 6.4, where contention-sensitive implementations of concurrent objects are presented.

6.1.3 The Case of Process Crashes

As already mentioned, if a process crashes while holding a lock, the processes that invoke a lock-based operation on the same object can be blocked forever. Hence, locks cannot cope with process crashes. This means that the implementations described in this chapter tolerate process crashes in all executions in which no process crashes while holding a lock.

6.2 A Static Hybrid Implementation of a Concurrent Set Object

This section presents a hybrid implementation of a concurrent set object due to S. Heller, M. Herlihy, M. Luchangco, M. Moir, W. Scherer, and N. Shavit (2007).

6.2.1 Definition and Assumptions

Definition A concurrent set object S is defined by the three following operations:

- S.add(v) adds v to the set S and returns *true* if v was not in the set. Otherwise it returns *false*.
- S.remove(v) suppresses v from S and returns *true* if v was in the set. Otherwise it returns *false*.
- S.contain(v) returns *true* if $v \in S$ and *false* otherwise.

Assumptions It is assumed that the elements that the set can contain belong to a well-founded set. This means that they can be compared, have a smallest element, have a greatest element, and that there is a finite number of elements between any two elements.

Despite the fact that the implementation has to be correct for any pattern of operation invocations, it is assumed that the number of invocations of contain() is much bigger than the number of invocations of add() and remove(). This application-related assumption is usually satisfied when the set represents dictionary-like shared data structures. This assumption motivates the fact that the implementation of contain() is required to be wait-free while the implementations of add() and remove() are only required to be deadlock-free.

6.2.2 Internal Representation and Operation Implementation

The internal representation of a set The set S is represented by a linked list pointed to by a pointer kept in an atomic register *HEAD*. A cell of the list (say *NEW_CELL*) is made up of four atomic registers:

- *NEW_CELL.val* which contains a value (element of the set).
- *NEW_CELL.out*, a Boolean (initialized to *false*) that is set to *true* when the corresponding element is suppressed from the list.
- *NEW_CELL.lock*, which is a lock used to ensure mutual exclusion (when needed) on the registers composing the cell. This lock is accessed with the operations acquire_lock() and release_lock().

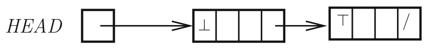


Fig. 6.1 The initial state of the list

- *NEW_CELL.next*, which is a pointer to the next cell. The set is organized as a sorted linked list. Initially the list is empty and contains two *sentinel* cells, as indicated in Fig. 6.1. The values associated with these cells are the default values denoted $\perp$ and $\top$. These values cannot belong to the set and are such that for any value v of the set we have $\perp < v < \top$. All operations are based on list traversal.

The algorithm implementing the operation S.remove(v) This algorithm is described in lines 1–9 of Fig. 6.2. Using the fact that the list is sorted in increasing order, the invoking process p_i traverses the list from the beginning until the first cell whose element v' is greater than v (lines 1–2). Then it locks two cells: the cell containing the element v' (which is pointed to by its local variable $curr$) and the immediately preceding cell (which is pointed to by its local variable $pred$).

The list traversal and the locking of the two consecutive cells are asynchronous, and other processes can concurrently access the list to add or remove elements. It is consequently possible that there are synchronization conflicts that make the content of $pred$ and $curr$ no longer valid. More specifically, the cell pointed to by $pred$ or $curr$ could have been removed, or new cells could have been inserted between the cells pointed to by $pred$ and $curr$. Hence, before suppressing the cell containing v (if any), p_i checks that $pred$ and $curr$ are still valid. The Boolean procedure validate($pred$, $curr$) is used to this end (lines 10–11).

If the validation predicate is false, p_i restarts the removal operation (line 9). This is the price that has to be paid to have an optimistic removal operation (there is no global locking of the whole list, which would prevent concurrent processes from traversing the list). Let us remember that, as by assumption there are few invocations of the remove() and add() operations, p_i will eventually terminate its invocation.

If the validation predicate is satisfied, p_i checks whether v belongs to the set or not (Boolean $pres$, line 5). If v is present, it is suppressed from the set (line 6). This is done in two steps:

- First the Boolean field *out* of the cell containing v is set to *true*. This is a *logical* removal (logical because the pointers have not yet been modified to suppress the cell from the list). This logical removal is denoted $S1$ in Fig. 6.3.
- Then, the *physical* removal occurs. The pointer $(pred \downarrow).next$ is updated to its new value, namely $(curr \downarrow).next$. This physical removal is denoted $S2$ in Fig. 6.3.

The algorithm implementing the operation S.add(v) This algorithm is described in lines 12–23 of Fig. 6.2. It is very close to the algorithm implementing the remove(v) operation. Process p_i first traverses the list until it reaches the cell whose value field is greater than v (lines 12–13) and then locks the cell that precedes it (line 14). Then, as previously, it checks if the values of its pointers $pred$ and $curr$ are valid (line 14). If they are valid and v is not in the list, p_i creates a new cell that contains v and inserts it into the list (lines 17–20).

```
operation $S$.remove($v$) is
 (1)  $pred \leftarrow HEAD$; $curr \leftarrow (HEAD \downarrow).next$;
 (2)  while $((curr \downarrow).val < v)$ do $pred \leftarrow curr$; $curr \leftarrow (curr \downarrow).next$ end while;
 (3)  $((pred \downarrow).lock)$.acquire_lock(); $((curr \downarrow).lock)$.acquire_lock(); $valid \leftarrow false$;
 (4)  if validate($pred, curr$)
 (5)     then $valid \leftarrow true$; $pres \leftarrow ((curr \downarrow).val = v)$;
 (6)          if $(pres)$ then $(curr \downarrow).out \leftarrow true$; $(pred \downarrow).next \leftarrow (curr \downarrow).next$ end if
 (7)  end if;
 (8)  $((pred \downarrow).lock)$.release_lock(); $((curr \downarrow).lock)$.release_lock();
 (9)  if $(valid)$ then return($pres$) else restart the operation end if
end operation.

predicate validate($pred, curr$) is
 (10) let $res = (\neg((pred \downarrow).out) \wedge \neg((curr \downarrow).out) \wedge ((pred \downarrow).next = curr))$;
 (11) return($res$)
end predicate.

operation $S$.add($v$) is
 (12) $pred \leftarrow HEAD$; $curr \leftarrow (HEAD \downarrow).next$;
 (13) while $((curr \downarrow).val < v)$ do $pred \leftarrow curr$; $curr \leftarrow (curr \downarrow).next$ end while;
 (14) $((pred \downarrow).lock)$.acquire_lock(); $valid \leftarrow false$;
 (15) if validate($pred, curr$)
 (16)    then $valid \leftarrow true$; $to\_add \leftarrow ((curr \downarrow).val \neq v)$;
 (17)         if $(to\_add)$ then $NEW\_CELL \leftarrow$ new_cell(); $NEW\_CELL.out \leftarrow false$;
 (18)                             $NEW\_CELL.val \leftarrow v$; $NEW\_CELL.next \leftarrow curr$;
 (19)                             $NEW\_CELL.lock \leftarrow open$; $(pred \downarrow).next \leftarrow (\uparrow new\_cell)$
 (20)         end if
 (21) end if;
 (22) $((pred \downarrow).lock)$.release_lock();
 (23) if $(valid)$ then return($to\_add$) else restart the operation end if
end operation.

operation $S$.contain($v$) is
 (24) $curr \leftarrow HEAD$;
 (25) while $((curr \downarrow).val < v)$ do $curr \leftarrow (curr \downarrow).next$ end while;
 (26) let $res = ((curr \downarrow).val = v) \wedge (\neg(curr \downarrow).out)$;
 (27) return($res$)
end operation.
```

Fig. 6.2 Hybrid implementation of a concurrent set object

It is interesting to observe that, as in the removal operation, the addition of a new element v is done in two steps. The field *NEW_CELL.next* is first updated (line 18). This is the *logical* addition (denoted $S1$ in Fig. 6.4). Only then, is the field $(pred \downarrow).next$ updated to a value pointing to *NEW_CELL* (line 18). This is the *physical* addition (denoted $S1$ in Fig. 6.4). (Let us notice that the lock associated with the new cell is initialized to the value *open*.)

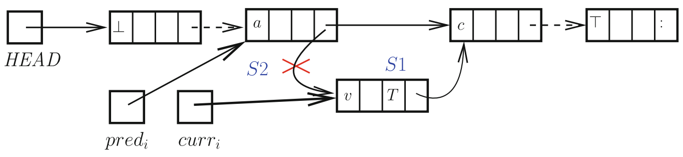


Fig. 6.3 The remove() operation

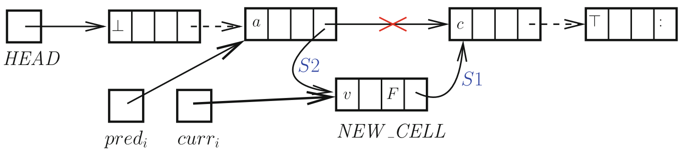


Fig. 6.4 The add() operation

Finally, p_i releases the lock on the cell pointed to by its local pointer variable ptr. It returns a Boolean value if the validation predicate was satisfied and restarts if it was not.

The algorithm implementing the operation S.contain(v) This algorithm is described in lines 24–24 of Fig. 6.2. As it does not use locks and cannot be delayed by locks used by the add() and remove() operations, it is wait-free. It consists of a simple traversal of the list. Let us remark that, during this traversal, the list does not necessarily remain constant: cells can be added or removed, and so the values of the pointers are not necessarily up to date when they are read by the process p_i that invoked S.contain(). Let us consider Fig. 6.5. It is possible that the pointer values $pred_i$ and $curr_i$ of the current invocation of contain(v) by p_i are as indicated in the figure while all the cells between those containing a_1 and b are removed (let us remark that it is also possible that a new cell containing the value v is concurrently added).

The list traversal is the same as for the add() and remove() operations. The value *true* is returned if and only if v is currently the value of the cell pointed to by $curr$ and this cell has not been logically removed. The algorithm relies on the fact that a cell cannot be recycled as long as it is reachable from a global or local pointer. (In contrast, cells that are no longer accessible can be recycled.)

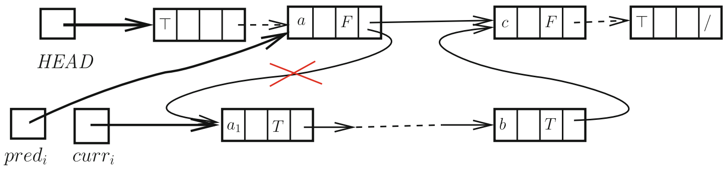


Fig. 6.5 The contain() operation

6.2.3 Properties of the Implementation

Base properties The previous implementation of a concurrent set has the following noteworthy features:

- The traversal of the list by an add()/remove() operation is wait-free (a cell locked by an add()/remove() does not prevent another add()/remove() from progressing until it locks a cell).
- Locks are used on at most two (consecutive) cells by an add()/remove() operation.
- Invocations of the add()/remove() operations on non-adjacent list entries do not interfere, thereby favoring concurrency.

Linearization points Let us remember that the linearization point of an operation invocation is a point of the time line such that the operation appears as if it was been executed instantaneously at that time instant. This point must lie between the start time and the end time of the operation.

The algorithm described in Fig. 6.2 provides the operations add(), remove(), and contain() with the following linearization points. Let an operation be *successful* (*unsuccessful*) if it returns *true* (*false*).

- remove() operation:
  - The linearization point of a successful remove(v) operation is when it marks the value v as being removed from the set, i.e., when it executes the statement $(curr \downarrow).out \leftarrow true$ (line 6).
  - The linearization point of an unsuccessful remove(v) operation is when, during its list traversal, it reads the first unmarked cell with a value $v' > v$ (line 2).
- add(v) operation:
  - The linearization point of a successful add(v) operation is when it updates the pointer $(pred \downarrow).next$ which, from then on, points to the new cell (line 19).
  - The linearization point of an unsuccessful add(v) operation is when it reads the value kept in $(curr \downarrow).val$ and that value is v (line 16).
- contain(v) operation:
  - The linearization point of a successful contain(v) operation is when it checks whether the value v kept in $(curr \downarrow).val$ belongs to the set, i.e., $(curr \downarrow).out$ is then false (line 26).
  - The linearization point of an unsuccessful contain(v) operation is more tricky to define. This is due to the fact that (as discussed previously with the help of Fig. 6.5), while contain(v) executes, an execution of add(v) or remove(v) can proceed concurrently.

    Let τ_1 be the time at which a cell containing v is found but its field *out* is marked *true* (line 26), or a cell containing $v' > v$ is found (line 25). Let τ_2 be the time just before the linearization point of a new operation add(v) that adds v to the set (if there is no such add(v), let $\tau_2 = +\infty$). The linearization point of an unsuccessful contain(v) operation is $\min(\tau_1, \tau_2)$.

The proof that this object construction is correct consists in (a) showing that any the operation contain() is wait-free and the operations of add() and remove() are deadlock-free, and (b) showing that, given any execution, the previous linearization points associated with the operation invocations define a trace that belongs to the sequential specification of the set object.

6.3 Contention-Sensitive Implementations

This section presents contention-sensitive implementations of two concurrent objects. The first object is a consensus object, while the second is a double-ended queue.

6.3.1 Contention-Sensitive Binary Consensus

Binary consensus object A consensus object provides a single operation denoted propose(v), where v is the value proposed by the invoking process. In a binary consensus object, only the values 0 and 1 can be proposed. An invocation of propose() returns a value which is said to be the value "decided" by the invoking process. A process can invoke the operation propose() at most once (hence, a consensus object is a one-shot object). Moreover any number of processes can invoke this operation. A process that invokes propose() is a *participating* process. The object is defined by the following three properties:

- Validity. A decided value is a proposed value.
- Agreement. No two processes decide different values.
- Termination. Any invocation of propose() terminates.

Favorable circumstances Here "favorable circumstances" (with respect to the contention-sensitiveness property of an implementation) concern two different cases. The first is when all the participating processes propose the same value. The second is when an invocation of propose() executes in a concurrency-free context. (Let us remember that two invocations prop_1 and prop_2 are not concurrent if prop_1 terminated before prop_2 started or prop_2 terminated before prop_1 started.)

When a favorable circumstance occurs, no lock has to be used. This means that an invocation of propose(v) is allowed to use the underlying locks only if (a) the other value $(1 - v)$ was previously or is currently proposed, and (b) there are concurrent invocations. Hence, from a lock point of view, the notion of conflict is related to both concurrency and proposed values.

Internal representation of a consensus object Let C be a consensus object. Its internal representation is made up of the following atomic read/write registers:

- *PROPOSED*[0..1], which is an array of two Boolean registers, both initialized to *false*. The atomic register *PROPOSED*[v] is set to *true* to indicate that a process has proposed value v.
- *DECIDED*, which is an atomic register whose domain is $\{\bot, 0, 1\}$. Initialized to $\bot$, it is eventually set to the value that is decided and never the value which is not decided.
- *AUX*, which is an atomic register whose domain and initial value are the same as for *DECIDED*. It can contain any value of its domain.
- *LOCK*, which is the starvation-free lock used to solve conflicts (if any).

The algorithm implementing the operation propose(v) This algorithm is described in Fig. 6.6. A process decides when it executes the statement return(val), where val is the value it decides. This algorithm is due to G. Taubenfeld (2009).

When a process p invokes propose(v), it first indicates that the value v was proposed and it writes v into *AUX* if this register is still equal to $\bot$ (line 1). Let us notice that, if several processes proposing different values concurrently read $\bot$ from *AUX*, each writes its proposed value in *AUX*.

Then, process p checks if the other binary value $(1 - v)$ was proposed by another process (line 2). If it is not the case, p writes v into *DECIDED* (line 3), and assuming that no other process has written a different value into *DECIDED* in the meantime, it decides the value stored in *DECIDED* (line 10). If the other value was proposed there is a conflict. Process p then decides the value kept in *DECIDED* if there is one (lines 4 and 10). If there is no decided value, the conflict is solved with the help of the lock (lines 5–7). Process p assigns the current value of *AUX* to *DECIDED* if that register was still equal to $\bot$ when it read it (lines 6) and p finally decides the value kept in *DECIDED*.

```
operation $C$.propose($v$) is
(1)  $PROPOSED[v] \leftarrow true$; if $(AUX = \bot)$ then $AUX \leftarrow v$ end if;
(2)  if $(\neg PROPOSED[1-v])$
(3)     then $DECIDED \leftarrow v$
(4)     else if $(DECIDED = \bot)$
(5)             then $LOCK$.acquire_lock();
(6)                  if $(DECIDED = \bot)$ then $DECIDED \leftarrow AUX$ end if;
(7)                  $LOCK$.release_lock()
(8)          end if;
(9)  end if;
(10) return($DECIDED$)
end operation.
```

Fig. 6.6 A contention sensitive implementation of a binary consensus object

Remark It is important to observe that this implementation of a binary consensus object uses only bounded registers and is independent of the number of processes that invoke the operation propose(v). The number of participating process can consequently be arbitrary (and even infinite).

Number of shared memory accesses An invocation which does not use the lock requires at most six accesses to atomic registers. Otherwise, it needs at most eight accesses to atomic registers plus two invocations of the lock. (One register access can be saved by saving v in a local variable if line 3 is executed, or by saving the value of $DECIDED \neq \bot$ read at line 6 if that line is executed.)

Theorem 21 *The algorithm described in Fig. 6.6 is a contention sensitive implementation of an atomic consensus object where "favorable circumstances" means the cases where all the invocations propose the same value or each invocation is executed in a concurrency free context.*

Proof A simple examination of the code of the algorithm shows that $DECIDED \neq \bot$ when a process decides at line 10. Moreover, the register $DECIDED$ can be assigned by a process only a proposed value (line 03) or the current value of AUX (line 6), but in the latter case, AUX has necessarily been previously assigned to a proposed value at line 6 by the same or another process. The consensus validity property follows.

The termination property follows immediately from the code if the invocation of C.propose() does not use the lock from the fact that the lock is starvation-free for the invocations which use it.

As far the agreement property is concerned, we show that a single value is written to the register $DECIDED$ (the same value can be written by several processes). We consider the two cases.

- Case 1. A single value is written in AUX (line 1). Let v be that value.
  As $1 - v$ is not written in AUX, we conclude that any process p_j that proposes $1 - v$ reads v from AUX at line 1 and consequently reads $PROPOSED[v] = true$ at line 2. It follows that p_j executes lines 4–8.

  Let p_i be any process that proposes v. If it finds $PROPOSED[1 - v] = false$ (line 2), it assigns v to $DECIDED$ (Fig. 6.7). Otherwise (similarly to process p_j) p_i executes the lines 4–8. If $DECIDED = \bot$ when either of p_i or p_j executes line 6, it assigns the current value of AUX (i.e., v) to $DECIDED$. Moreover, the processes that have proposed v and execute line 3 also assign v to $DECIDED$. Hence, $DECIDED$ is assigned the single value v, which proves the case.

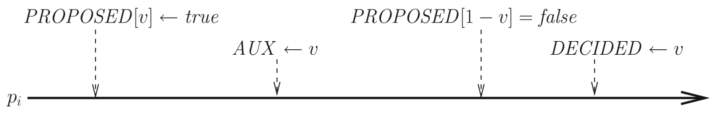


Fig. 6.7 Proof of the contention sensitive consensus algorithm (a)

- Case 2. Both values v and $1 - v$ are written into *AUX* (line 1).
  Let p_i be a process that proposes v and reads $\perp$ from *AUX*, and p_j a process that proposes $1 - v$ and reads $\perp$ from *AUX*. As both p_i and p_j have read $\perp$ from *AUX*, we conclude that, at line 3, both p_i and p_j have read *true* from *PROPOSED*$[1 - v]$ and *PROPOSED*$[v]$, respectively (Fig. 6.8). It follows that both of them execute lines 4–8.

  Let us now consider a process p_k that proposes a value w and reads a non-$\perp$ value from *AUX*. As it reads a non-$\perp$ value and both *PROPOSED*$[0]$ and *PROPOSED*$[1]$ were equal to *true* when it read them, it follows that p_k necessarily reads *true* from *PROPOSED*$[1 - w]$. Hence, it executes lines 4–8.

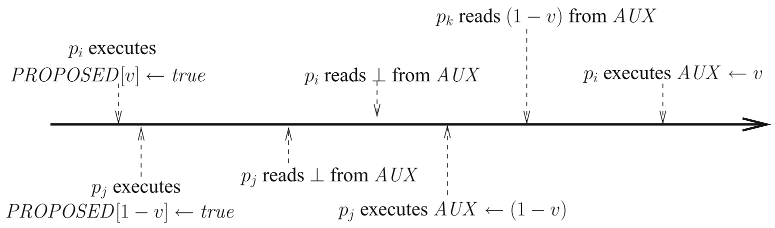
  It follows that all processes execute lines 4–8. The first process that acquires the lock writes the current value of *AUX* into *DECIDED*, and that value becomes the only decided value.

  (Let us notice that, due to the arbitrary speed of processes, it is not possible to predict if it is the first value written in *AUX* or the second one that will be the decided value.)

Let us now show that the implementation satisfies the contention sensitiveness property. We consider each case of "favorable circumstances" separately.

- Case 1: all participating processes propose the same value v.
  In this case, *PROPOSED*$[1 - v]$ remains forever equal to *false*. It follows that all the processes that invoke C.propose() write v into the atomic register *DECIDED* (line 3). Consequently none of the participating processes ever execute the lines 4–8, which proves the property.

- Case 2: the invocations of C.propose(v) are not concurrent.
  Let us consider such an invocation. If it is the first one, it writes v into *DECIDED* (line 3) and does not execute the lines 4–8, which proves the property. If other invocations have been executed before this one, they have all terminated and at least one of them has written a value into *DECIDED* (at line 3 or 6). Hence, the considered invocation C.propose(v) executes line 4, and as *DECIDED* $\neq \perp$, it does not execute lines 4–8, which concludes the proof of the contention sensitiveness property.



p_k reads $(1-v)$ from AUX
p_i executes
$PROPOSED[v] \leftarrow true$
p_i reads $\perp$ from AUX
p_i executes $AUX \leftarrow v$
p_j executes
$PROPOSED[1-v] \leftarrow true$
p_j reads $\perp$ from AUX
p_j executes $AUX \leftarrow (1-v)$

Fig. 6.8 Proof of the contention sensitive consensus algorithm (b)

Finally, as far as the atomicity property is concerned, the linearization point of an invocation of $C.\mathsf{propose}(v)$ is defined as follows:

- If the invoking process executes line 3, the linearization point of $C.\mathsf{propose}(v)$ is the linearization point of the underlying atomic write $DECIDED \leftarrow v$.
- If the invoking process executes line 4, the linearization point of $C.\mathsf{propose}(v)$ depends on the predicate ($DECIDED = \bot$) checked at line 6.
  - If $DECIDED \neq \bot$, it is the linearization point of this read of *DECIDED* (which returned a non-$\bot$ value).
  - If $DECIDED = \bot$, it is the linearization point of the write of a value w into *DECIDED* (where w is the value previously obtained from the atomic register *AUX*). □

6.3.2 A Contention Sensitive Non-blocking Double-Ended Queue

The double-ended queue A double-ended queue has two heads: one on its left side and one on its right side. The head on one side is the last element of the queue seen from the other side. Such an object has four operations:

- The operation right_enq(v) (or left_enq(v)) adds v to the queue such that v becomes the last value on the right (or left) side of the queue.
- The operation right_deq() (or left_deq()) suppresses the last element at the right (or left) of the queue. If the queue is empty, the operation returns the value *empty*.

A double-ended queue is defined by a sequential specification. This specification contains all the correct sequences including all or a subset of the operations. It follows that, in a concurrency context, a queue has to be an atomic object. A double-ended queue is a powerful object that generalizes queues and stacks. More precisely, we have the following (see Fig. 6.9, where the double-ended queue contains the list of values a, b, c, d, e, f):

- If either only the operations left_enq() and right_deq() or only the operations right_enq() and left_deq() are used, the object is a queue.
- If either only the operations left_enq() and left_deq() or only the operations right_enq() and right_deq() are used, the object is a stack.

Favorable circumstances The implementation that follows considers the following notion of "favorable circumstances" from the contention sensitiveness point of view: The operation invocations appear in a concurrency-free context. When this occurs, such an operation invocation is not allowed to use locks.

Internal representation of a double-ended queue Let DQ be a double-ended queue. Its internal representation is made up of the following objects:

- An infinite array $Q[-\infty..+\infty]$ of atomic registers whose aim is to contain the elements of the queue. Initially, all the registers $Q[x]$ such that $x < 0$ are initialized to a default control value denoted $\perp_\ell$, and all the registers $Q[x]$ such that $x \geq 0$ are initialized to a default control value denoted $\perp_r$.
- *LI* and *RI* are two atomic read/write registers that point to the first free entry when looking from the left side and the right side of the queue, respectively. *LI* is initialized to -1, while *RI* is initialized to 0.
- *L_LOCK* and *R_LOCK* are two locks. *L_LOCK* is used to solve conflicts (if any) between invocations of left_enq() and left_deq(). Similarly, *R_LOCK* is used to solve conflicts (if any) between invocations of right_enq() and right_deq().

The following invariant is associated with the internal representation of the double-ended queue:

$$\forall x, y : (x < y) \Rightarrow \big((Q[y] = \perp_\ell) \Rightarrow (Q[x] = \perp_\ell)\big) \wedge \big((Q[x] = \perp_r) \Rightarrow (Q[y] = \perp_r)\big).$$

Hence, at any time, the list of values which have been enqueued and not yet dequeued is the list kept in the array $Q[(LI+1)..(RI-1)]$. In Fig. 6.9, the current value of the double-ended queue is represented by the array $Q[-2..3]$.

Atomic operations for accessing a register $Q[x]$ An atomic register $Q[x]$ can be accessed by three atomic operations, denoted LL() (linked load), SC() (store conditional) and VL() (validate). These operations are provided by the hardware, and their effects are described by the algorithms of Fig. 6.10.

Let X be any register $Q[x]$. The description given in Fig. 6.10 assumes there are n processes whose indexes are in $\{1, \ldots, n\}$. It considers that a distinct Boolean array $VALID_X[1..n]$ is associated with each register X. This array is initialized to $[false, \ldots, false]$.

An invocation of X.LL() (linked load) returns the current value of X and links this read (issued by a process p_i) by setting $VALID_X[i]$ to *true* (line 1).

An invocation of X.SC$(-, v)$ (store conditional) by a process p_i is successful if no process has written X since p_i's last invocation of X.LL(). In that case, the write is executed (line 2) and the value *true* is returned (line 4). If it is not successful, the value *false* is returned (line 5). Moreover, if X.SC$(-, v)$ is successful, all the entries

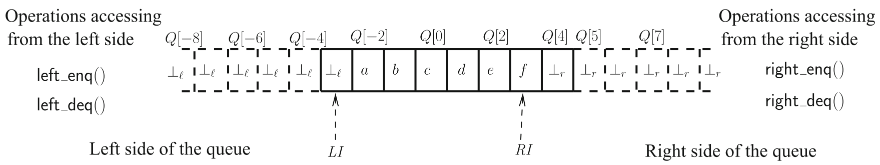


Fig. 6.9 A double-ended queue

```
operation $X$.LL($i$) is
(1)   $VALID_X[i] \leftarrow true$; return($X$)
end operation.

operation $X$.SC($i, v$) is
(2)   if ($VALID_X[i]$) then $X \leftarrow v$;
(3)                          for each $j \in [1..n]$ do $VALID_X[i] \leftarrow false$; end for
(4)                          return($true$)
(5)                     else return($false$)
(6)   end if
end operation.

operation $X$.VL($i$) is
(7)   return($VALID_X[i]$)
end operation.
```

Fig. 6.10 Definition of the atomic operations LL(), SC(), and VL() (code for process p_i)

of the array $VALID_X[1..n]$ are set to *false* (line 3) to prevent the processes that have previously invoked X.LL() from having a successful X.SC().

An invocation of X.VD() (validate) by a process p_i returns *true* if and only if no other process has issued a successful X.SC() operation since the last X.LL() invocation issued by p_i.

It is important to notice that between an invocation of X.LL() and an invocation of X.SC() or X.VL(), a process p_i can execute any code at any speed (including invocations of Y.LL(), Y.SC(), and Y.VL() where $Y \neq X$).

LL/SC primitives appear in MIPS architectures. Variants of these atomic operations are proposed in some architectures such as Alpha AXP (under the names idl_l and stl_c), IBM PowerPC (under the names lwarx and stwcx), or ARM (under the names ldrex and strex).

The algorithm implementing the operation right_enq(v) This algorithm is described in Fig. 6.11. A process p_i first saves the current value of *RI* in a local variable my_index (line 1). Then, p_i reads $Q[my\_index - 1]$ and $Q[my\_index]$ with the LL() atomic operation in order to both obtain their values and link these reads to their next (conditional) writes (line 2).

If these reads are such that $Q[my\_index - 1] \neq \bot_r$ and $Q[my\_index] = \bot_r$ (line 3), the right side of the queue (as defined by *RI*) has not been modified since p_i read *RI*. In that case, there is a chance that the right_enq(v) might succeed. Process p_i consequently checks if no process has modified $Q[my\_index - 1]$ since it read it. To that end, p_i executes $Q[my\_index - 1]$.SC($prev$) (line 4). If this conditional write (which does not change the value of $Q[my\_index - 1]$) is successful, p_i executes the conditional write $Q[my\_index]$.SC(v) to add the value v as the last element on the right of the double-ended queue (line 5). If no process has modified $Q[my\_index]$

```
operation $DQ$.right_enq($v$) is
(1)  $my\_index \leftarrow RI$;
(2)  $prev \leftarrow Q[my\_index-1]$.LL(); $cur \leftarrow Q[my\_index]$.LL();
(3)  if $(prev \neq \bot_r) \wedge (cur = \bot_r)$ then
(4)     if $Q[my\_index-1]$.SC($prev$) then
(5)        if $Q[my\_index]$.SC($v$) then $RI \leftarrow RI+1$; return($done$) end if
(6)     end if
(7)  end if;
(8)  $term \leftarrow false$;
(9)  $R\_LOCK$.acquire_lock();
(10)  repeat $my\_index \leftarrow RI$;
(11)          $prev \leftarrow Q[my\_index-1]$.LL(); $cur \leftarrow Q[my\_index]$.LL();
(12)          if $(prev \neq \bot_r) \wedge (cur = \bot_r)$ then
(13)             if $Q[my\_index-1]$.SC($prev$) then
(14)                if $Q[my\_index]$.SC($v$) then $RI \leftarrow RI+1$; $term \leftarrow true$ end if
(15)             end if
(16)          end if;
(17)  until ($term$) end repeat;
(18) $R\_LOCK$.release_lock();
(19) return($done$)
end operation.

operation $DQ$.right_deq() is
(20) $my\_index \leftarrow RI$;
(21) $prev \leftarrow Q[my\_index-1]$.LL(); $cur \leftarrow Q[my\_index]$.LL();
(22) if $(prev \neq \bot_r) \wedge (cur = \bot_r)$ then
(23)    if $(prev = \bot_\ell) \wedge Q[my\_index-1]$.VL() then return($empty$) end if;
(24)    if $Q[my\_index]$.SC($\bot_r$) then
(25)       if $Q[my\_index-1]$.SC($\bot_r$) then $RI \leftarrow RI-1$; return($prev$) end if
(26)    end if
(27) end if;
(28) $term \leftarrow false$;
(29) $R\_LOCK$.acquire_lock();
(30)  repeat $my\_index \leftarrow RI$;
(31)          $prev \leftarrow Q[my\_index-1]$.LL(); $cur \leftarrow Q[my\_index]$.LL();
(32)          if $(prev \neq \bot_r) \wedge (cur = \bot_r)$ then
(33)             if $(prev = \bot_\ell) \wedge Q[my\_index-1]$.VL() then return($empty$) end if;
(34)             if $Q[my\_index]$.SC($\bot_r$) then
(35)                if $Q[my\_index-1]$.SC($\bot_r$) then $RI \leftarrow RI-1$; $term \leftarrow true$ end if
(36)             end if
(37)          end if;
(38)  until ($term$) end repeat;
(39) $R\_LOCK$.release_lock();
(40) return($prev$)
end operation.
```

Fig. 6.11 Implementation of the operations right_enq() and right_deq() of a double-ended queue

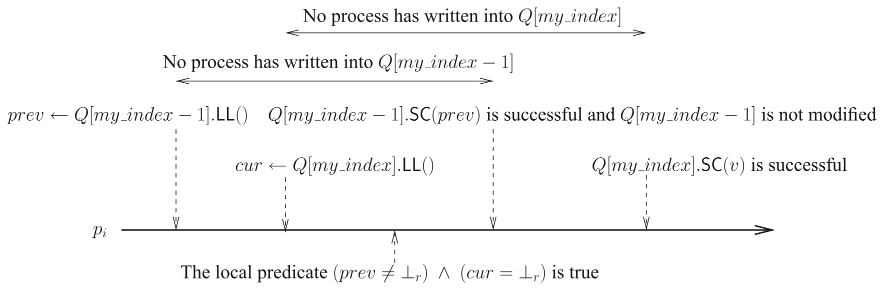

No process has written into $Q[my\_index]$
No process has written into $Q[my\_index-1]$
$prev \leftarrow Q[my\_index-1].\mathsf{LL}()$
$Q[my\_index-1].\mathsf{SC}(prev)$ is successful and $Q[my\_index-1]$ is not modified
$cur \leftarrow Q[my\_index].\mathsf{LL}()$
$Q[my\_index].\mathsf{SC}(v)$ is successful
p_i
The local predicate $(prev \neq \perp_r) \wedge (cur = \perp_r)$ is true

Fig. 6.12 How DQ.right_enq() enqueues a value

since it was read by p_i at line 2, the write is successful and p_i consequently increases the right index *RI* and returns the control value *done*. This behavior, which entails the enqueue of v on the right side, is described in Fig. 6.12.

If the previous invocations of LL() and SC() (issued at lines 2, 4, and 5) reveal that the right side of the double-ended queue was modified, p_i requires the lock in order to solve conflicts among the invocations of DQ.right_enq() (line 9). It then executes a loop in which it does the same as before. Lines 10–16 are exactly the same as lines 1–7 except for the statement return() at line 5, which is replaced at line 14 by $term \leftarrow true$ to indicate that the value v was added to the right side of the double-ended queue. When this occurs, the process p_i releases the lock and returns the control value *done*.

Let us consider an invocation of right_enq() (by a process p) which is about to terminate. More precisely, p starts executing the statements in the **then** part at line 5 (or line 14). If other processes are concurrently executing right_enq(), they will loop until p has updated the right index *RI* to $RI + 1$. This is due to the fact that p modifies $Q[RI]$ at line 5 (or line 14) before updating *RI*.

The algorithm implementing the operation right_deq() This algorithm is described in Fig. 6.11. It follows the same principles as and is similar to the algorithm implementing the operation right_enq(). The main differences are the following:

- The first difference is due to the fact that, while the double-ended queue is unbounded, it can become empty. This occurs when $(Q[RI-1] = \perp_\ell) \wedge (Q[RI] = \perp_r)$. This is checked by the use of the operations LL() and VL() and the predicates of lines 22–23 (or lines 32–33). When this occurs, the control value *empty* is returned (line 23 or 33).
- The second difference is that a data value has to be returned when the double-ended queue is not empty. This is the value kept in $Q[RI-1]$ which was saved in the local variable *prev* (line 21 or 31). Moreover, $Q[RI-1]$ has to be updated to $\perp_r$.

  The update is done at line 25 or 35 by a successful invocation of $Q[my\_index-1]$. $\mathsf{SC}(\perp_r)$.

While the right_enq() operation issues SC() invocations first on $Q[my\_index - 1]$ and then on $Q[my\_index]$ (lines 4–5 or lines 13–14), the right_deq() operation has to issue them in the opposite order, first on $Q[my\_index]$ and then on $Q[my\_index - 1]$ (lines 24–25 or lines 34–35). This is due to the fact that right_enq() writes (a value v) into $Q[my\_index]$ while right_enq() writes ($\perp_r$) into $Q[my\_index - 1]$.

The algorithms implementing the operations left_enq() **and** left_deq() These algorithms are similar to the algorithms implementing the right_enq() and right_deq() operations. The only modifications to be made to the previous algorithms are the following: replace *RI* by *LI*, replace *R_LOCK* by *L_LOCK*, replace each occurrence of $\perp_r$ by $\perp_\ell$, and replace the occurrence of $\perp_\ell$ at line 33 by $\perp_r$.

A left-side operation and a right-side operation can be concurrent and try to invoke an atomic SC() operation on the same register $R[x]$. In such a case, if one is unsuccessful, it is because the other one was successful. More generally, the construction is non-blocking.

6.4 The Notion of an Abortable Object

6.4.1 Concurrency-Abortable Object

Definition A *concurrency-abortable* (in short abortable) object is an object such that any invocation of any of its operations (a) returns after a bounded number of steps (shared memory accesses) and (b) is allowed to return the default value $\perp$ in presence of concurrency. (A similar notion was introduced in Sect. 2.1.6.) When an invocation returns $\perp$, we say that it aborted. When considering the object internal representation level, an operation invocation that aborts is similar to an invocation that has not been issued. Differently, an operation invoked in a concurrency-free pattern never aborts and behaves as defined by the specification of the object.

Examples of invocations that occur in a concurrency-free pattern are described in Fig. 6.13. There are three processes that have issued six operation invocations, denoted inv_x, $1 \leq x \leq 6$. The invocations inv_1 and inv_2 are concurrent, as are the invocations inv_4 and inv_5. On the contrary, each invocation inv_3 and inv_6 is executed in a concurrency-free pattern. For each of them, any other invocation either terminated before it started or started after it terminated. (The notion of concurrency-free pattern can be easily formally defined using the notion of a history introduced in Chap. 4.)

Example: a non-blocking abortable stack Let us consider the non-blocking implementation of a bounded stack presented in Fig. 5.10. A simple implementation of a stack that is both abortable and non-blocking can be easily obtained as follows. Instead of looping when a compare and swap statement fails (i.e., returns *false*), an operation returns $\perp$. The corresponding implementation is described in

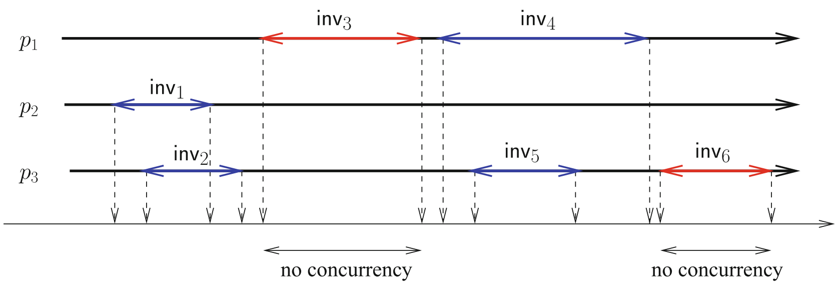


Fig. 6.13 Examples of concurrency-free patterns

Fig. 6.14, where the operations are denoted ab_push() and ab_pop(). The internal representation of the stack is the same as the one defined in Sect. 5.2.5 with the following simple modification: in each operation, the loops are suppressed and replaced by a return($\perp$) statement. It is easy to see that this modification does not alter the non-blocking property of the algorithm described in Fig. 5.10.

```
operation ab_push($v$) is
 (1)  $(index, value, seqnb) \leftarrow TOP$;
 (2)  help$(index, value, seqnb)$;
 (3)  if $(index = k)$ then return($full$) end if;
 (4)  $sn\_of\_next \leftarrow STACK[index+1].sn$;
 (5)  $newtop \leftarrow \langle index+1, v, sn\_of\_next+1\rangle$;
 (6)  if $TOP$.compare&swap$\big(\langle index, value, seqnb\rangle, newtop\big)$
 (7)          then return($done$) else return($\perp$) end if
end operation.

operation ab_pop() is
 (8)  $(index, value, seqnb) \leftarrow TOP$;
 (9)  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 $TOP$.compare&swap$\big(\langle index, value, seqnb\rangle, newtop\big)$
 (14)         then return($value$) else return($\perp$) end
end operation.

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

Fig. 6.14 A concurrency-abortable non-blocking stack

Abortion-related properties It is easy to see that (a) an invocation of the operation ab_push() or ab_pop() requires three or four shared memory accesses (three when the stack is full or empty and four otherwise) and (b) an invocation of ab_push() or ab_pop() that occurs in a concurrency-free pattern does not return $\perp$.

6.4.2 From a Non-blocking Abortable Object to a Starvation-Free Object

This section describes a simple contention sensitive algorithm which transforms the implementation of any non-blocking concurrency-abortable object into a wait-free implementation of the same object. This algorithm, which is based on a starvation-free lock, is described in Fig. 6.15. (Let us remember that a simple algorithm which builds a starvation-free lock from a deadlock-free lock was presented in Sect. 2.2.2.)

Favorable circumstances The "favorable circumstances" for this implementation of a starvation-free object occur each time an operation executes in a concurrency-free context. In such a case, the operation has to execute without using the underlying lock.

Notation The algorithm is presented in Fig. 6.15. Let oper(par) denote any operation of the considered object O and ab_oper(par) denote the corresponding operation on its non-blocking concurrency-abortable version *ABO*. This means that, when considering the stack object presented in the previous section, push() or pop() denote the non-abortable counterparts of ab_push() or ab_pop(), respectively. It is assumed that any invocation of an object operation oper(par) returns a value which is different from the default value $\perp$. As in the previous section, $\perp$ can be returned by invocations of ab_oper(par) only to indicate that they failed.

```
operation oper($par$) is
(1)   if ($\neg CONTENTION$)
(2)     then $res \leftarrow ABO$.ab_oper($par$); if ($res \neq \perp$) then return($res$) end if
(3)   end if;
(4)   $LOCK$.acquire_SF_lock();
(5)     $CONTENTION \leftarrow true$;
(6)     repeat $res \leftarrow ABO$.ab_oper($par$) until $res \neq \perp$ end repeat;
(7)     $CONTENTION \leftarrow false$;
(8)   $LOCK$.release_SF_lock();
(9)   return($res$)
end operation.
```

Fig. 6.15 From a concurrency-abortable object to a starvation-free object

Internal representation In addition to an underlying non-blocking abortable object *ABO*, the internal representation of the starvation-free object *O* is made up of the following objects;

- An atomic Boolean read/write register denoted *CONTENTION* which is initialized to *false*. This Boolean is used to indicate that there is a process that has acquired the lock and is invoking an underlying operation *ABO*.ab_oper(*par*).
- A starvation-free lock denoted *LOCK*. To insist on its starvation-free property, its operations are denoted acquire_SF_lock() and release_SF_lock().

The transformation The algorithm is described Fig. 6.15. When a process *p* invokes oper(*par*), it first checks if there is contention (line 1).

If $CONTENTION = true$, there are concurrent invocations and *p* proceeds to line 4 to benefit from the lock. If $CONTENTION = false$, *p* invokes *ABO*.ab_oper(*par*). As *ABO* is a concurrency-abortable object, this invocation (a) always terminates and (b) always returns a non-$\perp$ value *res* if there is no concurrent invocation. If this is the case, *p* returns that value *res*. On the contrary, if there are concurrent invocations, *ABO*.ab_oper(*par*) may return $\perp$. In that case, *p* proceeds to line 4 to compete for the lock.

LINES 4–9 constitute the lock-based part of the algorithm. When it has obtained the lock, a process first sets *CONTENTION* to *true* to indicate there is contention, and loops invoking the underlying operation *ABO*.ab_poper(*par*) until it obtains a non-$\perp$ value (i.e., its invocation is not aborted). When this occurs, it resets *CONTENTION* to *false* and releases the starvation-free lock.

Number of shared memory accesses It is easy to see that, in a concurrency-free context, an operation invocation requires a single shared memory access (to the register *CONTENTION*) in addition to the shared memory accesses involved in *ABO*.ab_oper(*par*). When the underlying object *ABO* is the stack described in the previous section, each invocation of *ABO*.ab_push(*par*) (or *ABO*.ab_pop()) requires three shared memory accesses if the stack is full (or empty), and four shared memory accesses otherwise.

In a concurrency context, an operation requires at most three accesses to the register *CONTENTION*, two accesses to the lock, and a finite but arbitrary number of accesses to the base objects which implement the underlying abortable object *ABO*.

Theorem 22 *Let us assume that the underlying lock is starvation-free and that the number of processes is bounded. Given a non-blocking abortable object ABO, the transformation described in Fig. 6.15 is a contention sensitive implementation of a starvation-free object O where "favorable circumstances" means each operation invocation is executed in a concurrency-free context. Moreover, if the underlying object ABO is atomic, so is the constructed object O.*

Proof Let us first consider the contention sensitiveness property. When an invocation of oper() executes in a concurrency-free context, the Boolean *CONTENTION* is equal to *false* when the operation starts and consequently oper() invokes *ABO*.ab_oper() at line 2. As *ABO* is a concurrency-abortable object and there are no concurrent operation invocations, the invocation of ab_oper() does not abort. It follows that this invocation of oper() returns at line 2, which proves the property.

Let us now show that the implementation is starvation-free, i.e., that any invocation of any operation oper() terminates. To this end, given an invocation inv_op$_p$ of an operation oper() issued by a process p, we have to show that there is eventually an underlying invocation of *ABO*.ab_oper() invoked by inv_op$_p$ that does not return $\perp$.

Let us first observe that, as *ABO* is a concurrency-abortable object, any invocation of *ABO*.ab_oper() terminates (returning $\perp$ or another value). If the underlying invocation *ABO*.ab_oper() issued at line 2 returns a non-$\perp$ value, inv_op$_p$ does terminate. If the underlying invocation *ABO*.ab_oper() returns $\perp$ or if the Boolean *CONTENTION* was equal to *false* when p executed line 1, p tries to acquire the lock (line 4).

Among the process that compete for the lock, let q be the process which has obtained and not yet released the lock. It repeatedly invokes some operation *ABO*.ab_oper$_q$() until it obtains a non-$\perp$ value. It is possible that other processes execute, concurrently with q, some underlying operations *ABO*.ab_oper1(), *ABO*.ab_oper2(), etc. This happens if these processes have found *CONTENTION* $=$ *false* at line 2 (which means that they have read *CONTENTION* before it was written by q). Hence, in the worst case, there are $(n-2)$ other processes executing operations on *ABO* concurrently with q (all the processes but p and q). As *ABO* is non-blocking, one of them returns a non-$\perp$ value and the corresponding process terminates its invocation of an operation on the underlying object *ABO*. If this process is q ,we are done. If it not q and invokes again an operation oper$'$(), it is directed to require the lock because now *CONTENTION* $=$ *false*.

Hence, there are now at most $(n - 3)$ processes executing operations on *ABO* concurrently with q. If follows that, if q has not obtained a non-$\perp$ value before, it eventually executes *ABO*.ab_poper$_q$() in a concurrency-free context. It then obtains a non-$\perp$ value and releases the lock.

As the lock is starvation-free, it follows that p eventually obtains it. Then, replacing q by p in the previous reasoning, it follows that p eventually obtains a non-$\perp$ value from an invocation of *ABO*.ab_oper() and accordingly terminates its upper-layer invocation of the operation oper().

The proof of atomicity follows from the following definition of the linearization points associated with the invocations of the underlying object *ABO*. Given an invocation of an operation O.oper(), let us consider its last invocation of *ABO*.ab_oper() (that invocation returned a non-$\perp$ value). The linearization point of oper() is the linearization of this underlying invocation. □

6.5 Summary

This chapter has introduced the notion of a hybrid implementation of concurrent objects and the notion of an abortable concurrent object.

Hybrid implementations are partly mutex-free and partly lock-based. Two kinds of hybridism have been presented: a static hybrid implementation considers mutex-free operations and lock-based operations, while a dynamic hybrid implementation is related to the current concurrency pattern (locks have not to be used in "favorables circumstances"). These notions have been illustrated by presenting a static hybrid implementation of a concurrent set object, and dynamic hybrid implementations of a consensus object and a double-ended queue. Interestingly, the notion of LL/SC base operations was introduced to implement the queue.

An abortable object is an object whose operation invocations can be aborted in the presence of concurrency. This notion has been illustrated with a non-blocking implementation of an abortable stack. Finally, it has been shown how a non-blocking abortable object can be transformed into a starvation-free object as soon as one can use a starvation-free lock.

6.6 Bibliographic Notes

- Without giving it the name "static hybrid", the notion of static hybrid implementation of a concurrent object was implicitly introduced by S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit in [137].

  The implementation of a concurrent set object described is Sect. 6.2 is due to to the same authors [137]. This implementation was formally proved correct in [78].

- The notion of contention sensitive implementation is due to G. Taubenfeld [263].

  The contention sensitive implementations of a binary consensus object and of a double-ended queue are due to G. Taubenfeld [263]. The second of these implementations is an adaptation of an implementation of a double-ended queue based on compare and swap() proposed by M. Herlihy, V. Luchangco, and M. Moir in [143] (the notion of obstruction-freedom is also introduced in this paper).

- The notion of concurrency-abortable implementation used in this chapter is from [214] where the methodology to go from a non-blocking abortable implementation to a starvation-free implementation of an object is presented. This methodology relies on a general approach introduced by G. Taubenfeld in [263].

- It is important to insist on the fact that the notion of "abortable object" used in this chapter is different from the one used in [16] (where an operation that returns $\perp$ may or may not have been executed).

6.7 Exercises and Problems

1. Compare the notions of obstruction-freedom and non-blocking with the notion of an abortable implementation. Are they equivalent? Are they incomparable? etc.

2. Design a contention sensitive implementation of a multi-valued consensus object where the number of different values that can be proposed is bounded and this bound b is known to the processes.

3. Considering only the successful invocations (i.e., the ones which do not return $\perp$), prove that the implementation of a non-blocking abortable stack described in Fig. 6.14 implements an atomic stack.

4. An implementation is k-contention sensitive if it uses locks only when the concurrency level surpasses k (i.e., when more than k operation invocations execute concurrently).

   Let us consider a system where the processes cooperate by accessing atomic read/write registers and atomic registers that can also be accessed by a swap() operation. (Let us remember that X.swap($local$), where X is an atomic register and $local$ a variable of the invoking process, exchanges atomically the content of X and $local$.)

   Considering such a system, design a 2-contention sensitive implementation of a binary consensus object. Such an implementation can be obtained by appropriate modifications of the algorithm presented in Fig. 6.6. Prove then that your implementation is correct.

   Solution in [263].

Chapter 7
Wait-Free Objects from Read/Write Registers Only

The two previous chapters were on the implementation of concurrent atomic objects (such as queues and stacks). More precisely, the aim of Chap. 5 was to introduce and illustrate the notion of a mutex-free implementation and associated progress conditions, namely obstruction-freedom, non-blocking and wait-freedom. The aim of Chap. 6 was to introduce and investigate the notion of a hybrid implementation. In both cases, the internal representation of the high-level object that was constructed was based on atomic read/write registers and more sophisticated registers accessed by stronger hardware-provided operations such as `compare&swap()`, `fetch&add()`, or `swap()`.

This chapter and the two following ones address another dimension when one is interested in building wait-free implementations of concurrent objects, namely the case where the only base objects that can be used are atomic read/write registers. Hence, these chapters investigate the power of base read/write registers to construct wait-free implementations. This chapter is on the wait-free implementation of weak counters and store-collect objects, while Chap. 8 addresses snapshot objects, and Chap. 9 focuses on renaming objects.

As we are concerned with wait-free implementations, let us remember that it is assumed that any number of processes may crash. Let us also remember that, as far as terminology is concerned, a process is *correct* in a run if it does not crash in that run; otherwise, it is faulty.

Keywords Adaptive implementation · Fast store-collect · Favorable circumstances · Infinitely many processes · Store-collect object · Weak counter

7.1 A Wait-Free Weak Counter for Infinitely Many Processes

This section has two aims: to present a wait-free implementation of a weak counter object and to show how to cope with an unknown and arbitrarily large number of processes. To that end, it first presents a very simple implementation of a (non-weak) counter and then focuses on the wait-free implementation of a weak counter that can be accessed by infinitely many processes.

7.1.1 A Simple Counter Object

A shared counter C is a concurrent object that has an integer value (initially 0) and provides the processes with two operations denoted increment() and get_count(). Informally, the operation increment() increases the value of the counter by 1, while the operation get_count() returns its current value. In a more precise way, the behavior of a counter is defined by the three following properties:

- Liveness. Any invocation of increment() or get_count() by a correct process terminates.
- Monotonicity. Let gt_1 and gt_2 be two invocations of get_count() such that (a) gt_1 returns c_1, gt_2 returns c_2, and gt_1 terminates before gt_2 starts. Then, $c_1 \leq c_2$.
- Freshness. Let gt be an invocation of get_count() and c the value it returns. Let c_a be the number of invocations of increment() that have terminated before gt starts and c_b be the number of invocations of increment() that have started before gt terminates. Then, $c_a \leq c \leq c_b$.

The liveness property expresses that the implementation has to be wait-free. Monotonicity and freshness are the safety properties which give meaning to the object; namely, they define the domain of the value returned by a get_count() invocation. As we will see in the proof of Theorem 23, the previous behavior can be defined by a sequential specification.

A simple implementation A concurrent counter can be easily built as soon as the number of processes n is known and the system provides one atomic SWMR read/write register per process. More precisely, let $REG[1..n]$ be an array of atomic registers initialized to 0, such that, for any i, $REG[i]$ can be read by any process but is written only by p_i.

The algorithm implementing the operations increment() and get_count() are trivial (Fig. 7.1). The invocation of increment() by p_i consists in adding 1 to $REG[i]$ ($local\_ct$ is a local variable of p_i, initialized to 0). The invocation of get_count() consists in reading (in any order) and summing up the values of all the entries of the array $REG[1..n]$.

Theorem 23 *The algorithms described in Fig. 7.1 are a wait-free implementation of an atomic counter object.*

Proof The fact that the operations are wait-free follows directly from their code. The proof that the construction provides an atomic counter is based on the atomicity of the underlying base registers. Let us associate a linearization point with each invocation as follows:

```
operation increment() is
    $local\_ct \leftarrow local\_ct + 1$; $REG[i] \leftarrow local\_ct$;
    return()
end operation.

operation get_count() is
    $r \leftarrow 0$;
    for $j \in \{1, \ldots, n\}$ do $r \leftarrow r + REG[j]$ end for;
    return($r$)
end operation.
```

Fig. 7.1 A simple wait-free counter for n processes (code for p_i)

- The linearization point associated with an invocation issued by a process p_i of the operation increment() is the linearization point of the underlying write operation of the underlying SWMR atomic register *REG*$[i]$. (If the invoking process p_i crashes before this write, it is as if the invocation had never been issued.)
- Let gt() be an invocation of the operation get_count() and c the value it returns. This invocation gt() is linearized at the time of the read of the underlying register *REG*$[x]$ such the value c is attained (i.e., the sum of the values obtained from the registers read before *REG*$[x]$ was strictly smaller than c and the registers read after it did not change the value of r).

  If several invocations return the same value c, they are ordered according to their start events (let us remember, see Part II, that no two operations are assumed to start simultaneously).

According to this definition of the linearization points, and the fact that no underlying atomic read/write register *REG*$[x]$ ever decreases and does increase each time p_x writes into *REG*$[x]$ when it executes the operation increment(), it is easy to conclude that (1) if two invocations of get_count() are sequential, the second one cannot return a value strictly smaller than the first one (monotonicity property), and (2) no invocation gt of the operation get_count() can return a value strictly smaller than the number of invocations of increment() that have terminated before tgt started or strictly greater than the number of invocations of increment() that have started before gt terminates (freshness property). □

7.1.2 Weak Counter Object for Infinitely Many Processes

Infinitely many processes This section focuses on dynamic systems where each run can have an unknown, arbitrarily large, and possibly infinite number of processes. The only constraint is that in each finite time interval only finitely many processes execute operations. Each process p_i has an identity i, and it is common knowledge that no two processes have the same identity.

Differently from the static model where there are n processes $p_1, \dots, p_n$, each process knowing n and the whole set of identities, now the identities of the processes that are in the system are not necessarily consecutive, and no process has a priori knowledge of which other processes can execute operations concurrently with it. (Intuitively, this means that a process can "enter" or "leave" the system at any time.) Moreover, no process is provided with an upper bound n on their number, which could be used by the algorithms (as, for example, in the previous algorithm, where the operation get_count() scans the whole array $REG[1..n]$). This model, called the *finite concurrency model*, captures existing physical systems where the only source of "infiniteness" is the passage of time.

It is important to see that the algorithms designed for this computation model have to be inherently wait-free as they have to guarantee progress even if new processes keep on arriving: the progress of pending operations cannot be indefinitely delayed because new processes keep on arriving.

Helping mechanism A basic principle when designing algorithms suited to the previous dynamic model with finite concurrency consists in using a helping mechanism. More generally, such mechanisms are central when one has to design wait-free implementations of concurrent objects.

More precisely, ensuring the wait-freedom property despite the fact that infinitely many processes can be involved in an algorithm requires a process to help other processes terminate their operations. This strategy prevents slow processes from never terminating despite the continuous arrival of new processes. This will clearly appear in the weak counter algorithms described below.

Weak counter: definition A *weak counter* is a counter whose increment() and get_count() operations satisfy the liveness and monotonicity properties of a classical counter (as defined previously), plus the following property (which replaces the previous freshness property):

- Weak increment. Let gt_1 and gt_2 be two invocations of the get_count() operation that return c_1 and c_2, respectively. Let incr be an invocation of increment() that (a) has started after gt_1 has terminated (i.e., $res[\text{gt}_1] <_H inv[\text{incr}]$ using the notations defined in Chap. 4), and (b) has terminated before gt_2 has started (i.e., $res[\text{incr}] <_H inv[\text{gt}_2]$). We have then $c_1 < c_2$.

With a classical counter, each invocation of the increment() operation, be it concurrent with other invocations or not, results in adding 1 to the value of the counter (if the invoking process does not crash before updating the SWMR register it is associated with). The way the counter increases is different for a weak counter. Let k be the number of concurrent invocations of increment() at some time. This concurrency pattern entails the increase of the counter by a quantity x such that $1 \le x \le k$. This means that the effect of a batch of k concurrent invocations of the operation increment() appears as being reduced to any number y, $1 \le y \le k$, of invocations, $k - y$ of these invocations being overwritten by other ones. No increment is missed only when there are no concurrent increment operations. It is easy to see that a counter is a weak counter but not vice versa. Moreover, differently from a counter, a weak counter has no sequential specification.

7.1.3 A One-Shot Weak Counter Wait-Free Algorithm

This section and the next one present an incremental construction of a weak counter object in a finite concurrency computation model. More precisely, this section presents and proves correct a wait-free implementation of a one-shot weak counter which is based on read/write registers only. *One-shot* means here that a process issues at most operation invocation. The next section will remove this restriction.

This construction of a wait-free weak counter is due to M.K. Aguilera (2004). Its main design principles are simplicity (more efficient constructions can be designed) and generality (in the sense that these principles can be used to obtain wait-free implementations of other objects in the presence of finite concurrency).

Internal representation of the weak counter object This internal representation consists of three arrays of atomic read/write registers. The first represents the counter itself, while the last two are used to implement the helping mechanism that provides processes with the wait-freedom property.

- The array *BIT* is made up of a potentially infinite number of MRMW registers. Each register $BIT[x]$ is initialized to the value 0 and can be set to the value 1 (by one or several processes). Its aim is to contain the value of the counter. More precisely, when the value of the counter is v, we have $BIT[x] = 1$ for $1 \leq x \leq v$, and $BIT[x] = 0$ for $x > v$.
- *READING* is a potentially infinite array of SWMR registers with one entry per process. $READING[i] = true$ means that the process p_i is currently trying to read the value of the counter and may require help to terminate its $get\_count()$ operation.
- *HELPED* is a potentially infinite array of MWSR registers with one entry per process. Its meaning is the following: $HELPED[i] = v$ means that some process helped p_i by providing it with the value v that p_i can use as the current value of the counter.

The algorithms implementing the operations increment() **and** get_count() These algorithms are described in Fig. 7.2. The algorithm implementing the increment() operation is fairly simple. When a process wants to increment the counter, it first obtains its current value v, and then sets to 1 the next bit of the array representing the value of the counter, namely $BIT[v + 1]$. (As we can see, no entry of the *BIT* array is reserved for some processes.)

The algorithm implementing the get_count() operation lies at the core of the construction. In addition to the helping arrays *READING* and *HELPED*, each invocation of get_count() by a process p_i uses a local index k, and a local variable to_help which will contain the identities of the processes that p_i can help terminate their get_count() invocation. The algorithm is made up of two parts:

```
operation increment() is
(1)  $v \leftarrow$ get_count(); $BIT[v+1] \leftarrow 1$; return()
end operation.

operation get_count() is
(2)  $k \leftarrow 1$; $to\_help \leftarrow \emptyset$;
(3)  $READING[i] \leftarrow true$;
(4)  while $\big((BIT[k]=1) \wedge (HELPED[i]=0)\big)$ do
(5)      if $(READING[k])$ then $to\_help \leftarrow to\_help \cup \{k\}$ end if;
(6)      $k \leftarrow k+1$
(7)  end while;
(8)  $READING[i] \leftarrow false$;
(9)  if $(HELPED[i] \neq 0)$ then return($HELPED[i]$)
(10)                  else let $v = k-1$;
(11)                       for each $j \in to\_help$, in the increasing order on $j$ do
(12)                            $HELPED[j] \leftarrow v$ end for
(13)                       return($v$)
(14) end if
end operation.
```

Fig. 7.2 Wait-free weak counter (one-shot version, code for p_i)

- The first part (lines 2–8) is a scan whose aim is to obtain the value of the counter.

  A process p_i starts scanning the *BIT* array until either it finds an entry equal to 0 or it discovers that it was helped by another process (line 4). During this scanning, p_i indicates to the other processes that it needs help by setting *READING*$[i]$ appropriately (lines 3 and 8). Moreover, it also registers the processes it could help before terminating (line 5).

- In the second part (lines 9–14) a process p_i returns a value. If it was helped by another process (we have then *HELPED*$[i] \neq 0$ at line 9), p_i returns the helping value supplied by another process. In that case, p_i does not help other processes. On the contrary, if it has not been helped, p_i computes the value v to be returned (line 10) and helps the processes in *to_help* with the value v it has obtained (lines 11–12) before returning v (line 13).

7.1.4 Proof of the One-Shot Implementation

As all the base read/write registers are atomic, it is possible to reason by considering a global time frame defined by the linearization points associated with the read and write operations on these registers. (This is the sequence $\widehat{S}$ as defined in Chap. 4 from the linearization points.) Let $f(\tau)$ denote the number of 1s in the array *BIT* at time τ.

Lemma 3 *The function $f(\tau)$ is non-decreasing.*

Proof The lemma follows directly from a simple examination of the algorithm: no entry of the array *BIT* is ever changed from 1 to 0. □

Lemma 4 *Let v be the value returned by* get_count() *at time τ. $\forall x : 1 \le x \le v$: $BIT[x] = 1$ at time τ.*

Proof Let v be the value returned by an invocation of get_count() issued by a process p_i that returns at time τ. There are two cases according to the line at which p_i returns:

- p_i returns at line 13. In this case, p_i exited the **while** loop with $HELPED[i] = 0$. The fact that the assertion $\forall x : 1 \le x \le v : BIT[x] = 1$ is satisfied at time τ follows from the following observations: (1) k is initialized to 1 and is increased in step of 1 (lines 2 and 6), (2) p_i exited the loop the first time it read 0 from $BIT[k]$, and (3) no entry of *BIT* is changed from 1 to 0.
- p_i returns at line 9. In this case, p_i returns because it found $HELPED[i] = v \neq 0$. This means that there is a process p_j that set $HELPED[i]$ to v (line 12) at some time $\tau' < \tau$. Due to the first item applied to p_j, the assertion $\forall x : 1 \le x \le v : BIT[x] = 1$ is true at time τ'. As $\tau' < \tau$ and no bit is reset to 0 after having been set to 1, it follows that the assertion is still satisfied at time τ'. □

Lemma 5 $(BIT[y] = 1) \Rightarrow (\forall x : 1 \le x \le y : BIT[x] = 1)$.

Proof An atomic read/write register $BIT[y]$ is set to 1 only in the algorithm implementing the operation increment(), and this is done just after an internal get_count() invocation which has returned $y - 1$ (line 1). Due to Lemma 4, all the entries $BIT[x]$ such that $1 \le x \le y - 1$ are then equal to 1. □

The following corollary is an immediate consequence of the previous lemmas.

Corollary 3 *At any time τ, the $f(\tau)$ entries of the array BIT that are equal to 1 are $BIT[1]$,..., $BIT[f(\tau)]$.*

Lemma 6 *Considering a process p_i that computes the value v at line 10 of the* get_count() *operation, let τ be the time at which it has read $BIT[k] = 0$ before exiting the **while** loop (line 7). We have $v = f(\tau)$.*

Proof As p_i executes line 10 and τ is the time at which it read $BIT[k] = 0$ before exiting the **while** loop, it follows from Corollary 3 and the fact that, at time τ, $BIT[k-1] = 1$ and $BIT[k] = 0$ that we have $f(\tau) = k - 1$ and consequently $v = f(\tau)$. □

Lemma 7 *Let* gt *be an invocation of the* get_count() *operation by a process p_i that starts at time τ_1 and terminates at time τ_2. The value v it returns is such that $f(\tau_1) \le v \le f(\tau_2)$.*

Proof Let us first show that $v \leq f(\tau_2)$. As exactly $f(\tau_2)$ entries of *BIT* are equal to 1 at time τ_2 and $f(\tau)$ is non-decreasing (Lemma 3), no invocation of the get_count() operation that exits the **while** loop at time $\tau \leq \tau_2$ can find $x > f(\tau_2)$ entries equal to 1. As the value v returned by gt is a value determined by an invocation of get_count() that exits the **while** loop before τ_2, the assertion $v \leq f(\tau_2)$ follows.

Let us now show that $f(\tau_1) \leq v$. Let τ' be the time at which p_i exits the **while** loop. We have $\tau_1 < \tau' < \tau_2$. There are two cases according to the line at which p_i returns its value:

- p_i returns at line 13. In this case, due to Lemma 6 we have $v = f(\tau')$. As, $f(\tau_1) \leq f(\tau')$ due to Lemma 3, we obtain $f(\tau_1) \leq v$, which proves the case.
- p_i returns at line 9. In this case, p_i returns the value v such that $v = \mathit{HELPED}[i]$. Let p_j be the process that updated $\mathit{HELPED}[i]$ to v. Let us consider the following time instants:

$$\begin{aligned}
\tau^i &= \text{time at which } p_i \text{ executes } \mathit{READING}[i] \leftarrow \mathit{true},\\
\tau_1^j &= \text{time at which } p_j \text{ reads } \mathit{READING}[i] = \mathit{true},\\
\tau_2^j &= \text{time at which } p_j \text{ executes } \mathit{to\_help} \leftarrow \mathit{to\_help} \cup \{i\},\\
\tau_3^j &= \text{time at which } p_j \text{ reads } \mathit{BIT}[k] = 0 \text{ and leaves the } \textbf{while}\\
&\quad\ \text{loop } (v \text{ is determined}),\\
\tau_4^j &= \text{time at which } p_j \text{ executes } \mathit{HELPED}[i] \leftarrow v.
\end{aligned}$$

We have $\tau_1 < \tau^i < \tau_1^j < \tau_2^j < \tau_3^j < \tau_4^j < \tau'$. Due to Lemma 3, we have $f(\tau_1) \leq f(\tau_3^j)$, and, due to Lemma 6, we have $v = f(\tau_3^j)$. If follows that $f(\tau_1) \leq v$. □

Theorem 24 (Monotonicity) *Let* gt_1 *and* gt_2 *be two invocations of* get_count() *such that* gt_1 *returns* v_1*,* gt_2 *returns* v_2*, and* gt_1 *terminates before* gt_2 *starts. Then,* $v_1 \leq v_2$.

Proof Let τ_1 be the time at which gt_1 terminates and τ_2 be the time at which gt_2 starts. Due to Lemma 7 we have $v_1 \leq f(\tau_1)$ and $f(\tau_2) \leq v_2$. Due to Lemma 3, we have $f(\tau_1) \leq f(\tau_2)$. Combining these inequalities results in $v_1 \leq v_2$. □

Lemma 8 *Let us consider an invocation of the* increment() *operation that starts at time* τ_1 *and terminates at time* τ_2*. We have* $f(\tau_1) + 1 \leq f(\tau_2)$.

Proof Noticing that the increment() operation issues an internal call to the get_count() operation, let τ_3 and τ_4 be the time instants at which this invocation of get_count() starts and terminates, respectively. We have $\tau_1 < \tau_3 < \tau_4 < \tau_2$, and consequently (due to Lemma 3) we also have $f(\tau_1) \leq f(\tau_3)$.

Let v the value returned by get_count(). Due to Lemma 7 we have $f(\tau_3) \leq v$. Moreover, due to the second line of the increment() operation (namely, *BIT* $[v + 1] \leftarrow 1$) and Corollary 3, we have $f(\tau_2) \geq v + 1$. Combining the previous

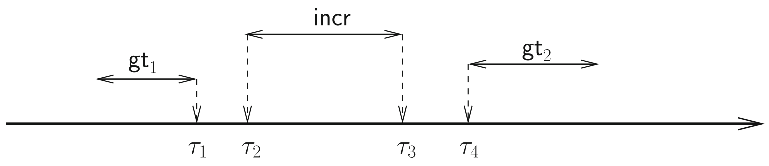


Fig. 7.3 Proof of the weak increment property

inequalities, we obtain $f(\tau_1) \leq f(\tau_3) \leq v < v+1 \leq f(\tau_2)$, which proves the lemma. □

The previous lemma captures the effect of the invocations of the increment() operation on the value of the weak counter. It states that, in the presence of one or several concurrent invocations of that operation, the counter is increased by at least 1, whatever the number of concurrent invocations. It does not state that the counter is increased by c if there are c concurrent invocations of the increment() operation.

Theorem 25 (Weak increment) *Let* gt$_1$ *and* gt$_2$ *be two invocations of* get_count() *that return* v_1 *and* v_2*, respectively, and* incr *an invocation of* increment() *such that* gt$_1$ *terminates before* incr *starts and* gt$_2$ *starts after* incr *terminates. We have* $v_1 < v_2$.

Proof Let the time instants be as defined in Fig. 7.3. We have $\tau_1 < \tau_2 < \tau_3 < \tau_4$. Moreover, we have the following inequalities:

$$\begin{aligned}
v_1 &\leq f(\tau_1) && \text{(Lemma 7)},\\
f(\tau_1) &\leq f(\tau_2) && \text{(Lemma 3)},\\
f(\tau_2) &< f(\tau_3) && \text{(Lemma 8)},\\
f(\tau_3) &\leq f(\tau_4) && \text{(Lemma 3)},\\
f(\tau_4) &\leq v_2 && \text{(Lemma 7)},
\end{aligned}$$

from which we easily conclude $v_1 < v_2$. □

Theorem 26 (Wait-freedom property) *Any invocation of an operation issued by a correct process terminates.*

Proof As any invocation of the increment() operation has no loop and invokes the get_count() operation, we only have to show that any invocation of the get_count() operation issued by a correct process always terminates. The proof is by contradiction.

Let us assume that a correct process p_i invokes get_count() and this invocation never terminates. Due to the code of get_count() (Fig. 7.2), we conclude that p_i loops forever in the **while** loop (lines 4–7) or in the **for** loop (lines 11–12).

Let us first consider the case of the **for** loop. If p_i loops forever in that loop, its local set *to_help* contains an infinity of process identities, which means that it has executed line 5 an infinite number of times. It follows that p_i never exited the **while** loop which contradicts the fact that it is executing the **for** loop. It follows that, if a process executes the **for** loop, its local set contains a finite number of identities. Hence, no process can loop forever in the **for** loop.

Let us now consider the case of the **while** loop. If p_i loops forever in that loop, $HELPED[i]$ remains equal to 0 and, for every k, p_i reads $BIT[k] = 1$. It follows that there are infinitely many invocations of increment() that terminate (the corresponding process having crashed after updating some $BIT[k]$ to 1). As each of these invocations of increment() calls get_count(), there are infinitely many invocations of get_count() that terminate. We consider two (non-exclusive) cases:

- Infinitely many invocations of get_count() terminate at line 13.
  In this case, as at any time the number of operation invocations which are currently executing is finite (model assumption), there is a process p_j that has invoked get_count() (and –due the case assumption– this invocation terminates at line 13) after p_i has invoked its (non-terminating) get_count() operation. In such a setting, p_j reads $READING[i] = true$ (line 5) and, consequently, adds i to its local variable to_help. Then (due to the case assumption) p_j executes lines 10-12, and sets $HELPED[i]$ to a value $v \neq 0$. After this has been done, p_i eventually exits its **while** loop, contradicting the initial assumption.

- Infinitely many invocations of get_count() terminate at line 9.
  In this case, due to the one-shot assumption, infinitely many processes p_j with identity $j > i$ are such that $HELPED[j]$ was set to a value different from 0. Each such p_j is helped by at least one process $p_{j'}$ (that sets $HELPED[j]$ to a non-0 value at line 12). As (1) there are infinitely many processes p_j, (2) a process executes only one operation invocation, and (3) at any time the number of current operation invocations is finite, it follows that there are infinitely many helping processes $p_{j'}$. (Let us notice that, for each p_j, it is possible that the processes that help it crash just after having updated $HELPED[j]$ and before helping other processes.) Consequently at least one of the helping processes (say p_g) has started its invocation of get_count() after p_i started its non-terminating invocation of get_count(). It follows that p_g reads $READING[i] = true$ and consequently adds i to its local variable to_help.

  As p_g sets $HELPED[j]$ to a non-0 value v, it has not crashed before. Moreover, as it executes the loop of lines 11–12 according to the increasing order of process identities that are in its local variable to_help, it follows from the fact that $i < j$ that p_g has updated $HELPED[i]$ to the value v before updating $HELPED[j]$ to v. After this has been done, p_i eventually exits its **while** loop, contradicting the initial assumption. □

Remark From an understanding point of view, it is important to notice that wait-freedom of the implementation is the only property whose proof relies on both the one-shot assumption and the order on process identities.

```
operation fast_get_count() is
(1) $k \leftarrow SHORTCUT$;
(2) $READING[i] \leftarrow true$;
(3) while $\big((BIT[k] = 1) \wedge (HELPED[i] = 0)\big)$ do $k \leftarrow k + 1$ end while;
(4) $READING[i] \leftarrow false$;
(5) if $(HELPED[i] \neq 0)$ then return$(HELPED[i])$
(6)                               else let $v = k - 1$; $SHORTCUT \leftarrow v$; return$(v)$
(7) end if
end operation.
```

Fig. 7.4 Fast read of a weak counter (code for process p_i)

7.1.5 A Multi-Shot Weak Counter Wait-Free Algorithm

This section presents two improvements of the previous implementation of a one-shot weak counter. It first presents a fast read operation that returns a correct value of the weak counter, and then removes the "one-shot" restriction.

Adding a fast_get_count() **operation** In the previous construction, each get_count() operation scans the *BIT* array from its very first entry *BIT*[1]. It is possible to avoid scanning all the bits, keeping the last scanned bit equal to 1 in an atomic register *SHORTCUT* (initialized to 1). We thus obtain the fast_get_count() algorithm described in Fig. 7.4. This algorithm does not allow a process to help other processes.

The operations that are now exported to the user are increment() (Fig. 7.2) and fast_get_count() (Fig. 7.4). But the implementation still uses the algorithm get_count() (Fig. 7.2), which is now an internal procedure called by the increment() operation. This is required in order for the helping mechanism to work.

An invocation of fast_get_count() cannot prevent another invocation of fast_get_count() from terminating. Actually, what can prevent an invocation of fast_get_count() from terminating are invocations of increment() (if there is no increment of the counter, all invocations trivially terminate). Hence, only the invocations of increment() have to help the other invocations.

Let us finally notice that, if invocations of the operation increment() stop during a long enough period, there is a time in that period after which the invocations of fast_get_count() take constant time.

Removing the one-shot restriction This section shows how to remove the one-shot restriction of the implementation described in Fig. 7.2. Extending it to the fast_get_count() operation is left as an exercise.

To remove the one-shot restriction we need to prevent a process p_i from obtaining an old help value from a slow process p_j that is currently helping one of its previous invocations of the get_count() operation. One way to fix this problem consists in associating an identity (a pair $\langle i, x\rangle$, where i is a process identity and x a sequence

```
operation get_count() is
(1)  $LAST\_INV[i] \leftarrow LAST\_INV[i] + 1$; let $x = LAST\_INV[i]$;
(2)  $k \leftarrow 1$; $to\_help \leftarrow \emptyset$;
(3)  $READING[i, x] \leftarrow true$;
(4)  while $\big((BIT[k] = 1) \wedge (HELPED[i, x] = 0)\big)$ do
(5)      let $y = LAST\_INV[k]$;
(6)      if $(READING[k, y])$ then $to\_help \leftarrow to\_help \cup \{\langle k, y\rangle\}$ end if
(7)  end while;
(8)  $READING[i, x] \leftarrow false$;
(9)  if $(HELPED[i, x] \neq 0)$ then return$(HELPED[i, x])$
(10)                          else let $v = k - 1$;
(11)                               foreach $\langle j, z\rangle \in to\_help$, in the increasing order defined by $j + z$ do
(12)                                   if $(LAST\_INV[j] = z)$ then $HELPED[j, z] \leftarrow v$ end if end do
(13)                               return$(v)$
(14) end if
end operation.
```

Fig. 7.5 Reading a weak counter (non-restricted version, code for process p_i)

number) with each invocation of get_count() and enriching the shared data structures as follows:

- An SWMR atomic register *LAST_INV*$[i]$ (initialized to 0) is associated with each process p_i. This register is used by p_i to generate sequence numbers.
- The atomic registers *READING*$[i]$ and *HELPED*$[i]$ now become two-dimensional arrays. Let us consider the xth invocation of get_count() issued by p_i.
  - *READING*$[i, x] = $ *true* means that p_i is looking for a counter value for its xth invocation.
  - *HELPED*$[i, x]$ is destined to contain the help value for its its xth invocation.

The wait-free property can be prevented only in runs where there are infinitely many invocations of the get_count() operation. This can occur when there are infinitely many processes that invoke get_count() or when a process invokes infinitely many get_count(). This observation can be used as follows by the helping mechanism in order to obtain a wait-free algorithm. A process p_i is required to help the invocations of get_count() whose identities $\langle j, z\rangle$ have been collected in its set *to_help*, in the order defined by the sum $j + z$. This order relation allows replacing, in the proof of the second item of Theorem 26, the order on the process identities by an order on invocations.

The corresponding general get_count() algorithm is described in Fig. 7.5. It is a straightforward extension of the base one-shot algorithm. The increment() operation is the same as before.

The proof of the general construction is left as an exercise. We consider here only the proof of the wait-freedom property.

Theorem 27 *Any operation issued by a correct process terminates.*

Proof The proof is nearly the same as that of Theorem 26, being made up of two items. The first item considers the case where infinitely many get_count() operations terminate without being helped by other operations. That proof still works, after having replaced $READING[i]$ and $HELPED[i]$ by $READING[i, x]$ and $HELPED[i, x]$, respectively.

The proof of the second item (namely, the case where there are infinitely many get_count() operations that terminate at line 9 due to the help of other operations) has to be adapted to the new setting, as now a process can sequentially issue as many operations as it wants.

Let $\langle i, x\rangle$ be the identity of an invocation of get_count() that never terminates. Due to the case assumption (infinitely many get_count() terminate at line 9), it follows that infinitely many invocations $\langle j, y\rangle$ with $i+x < j+y$ are such that $HELPED[j, y]$ was set to a non-0 value. Each such operation was helped by at least one operation $\langle j', y'\rangle$. As there are infinitely many invocations $\langle j, y\rangle$ and at any time the number of concurrent operations is finite, there are infinitely many helping invocations $\langle j', y'\rangle$. Hence, at least one of these helping invocations $\langle g, z\rangle$ started after the invocation $\langle i, x\rangle$ helped an invocation $\langle j'', y''\rangle$ such that $i + x < j'' + y''$. Consequently, the invocation $\langle g, z\rangle$ found $READING[i, x] = true$ and added $\langle i, x\rangle$ to to_help. As (1) $\langle g, z\rangle$ executes the loop defined at lines 11–12, (2) $\langle g, z\rangle$ helps $\langle j'', y''\rangle$, and (3) $i + x < j'' + y''$, it follows that $\langle g, z\rangle$ helps $\langle i, x\rangle$, contradicting the fact that $\langle i, x\rangle$ does not terminate. □

7.2 Store-Collect Object

This section introduces the notion of a *store-collect* object and presents several wait-free implementations of it in a classical static system made up of n processes $p_1, \ldots, p_n$.

7.2.1 Store-Collect Object: Definition

Informal description In a lot of applications, processes cooperate in the following way. Repeatedly, after it has computed a new value, a process deposits it in a shared data structure so that the other processes can read and use it. The last value deposited by a process overwrites the value it previously deposited (if any). Hence, a process makes public one value that it can update from time to time. Moreover, the processes progress asynchronously, each process computing and depositing values at its own pace.

This cooperation mechanism is abstracted in what is called a *store-collect* object. A process uses the operation store() to deposit a value, and the operation collect() to obtain the last values (one per process) that have been deposited. These operations are wait-free: they always terminate when invoked by a correct process.

These values define what is sometimes called a *view*. More precisely, a view is a set of pairs (process identity, value) with at most one pair per process. Initially, the view associated with a store-collect object is empty.

Partial order on views An invocation of the operation collect() returns a view containing the latest values made public by each process. To define precisely the notion of "latest values" returned in a view, we use a partial order relation defined on views. Let $view_1$ and $view_2$ be two views. $view_1 \leq view_2$ if, for every process p_i such that $(i, v1) \in view_1$, we have $(i, v2) \in view_2$ where the invocation of store($v2$) follows (or is) the operation $store(v1)$ (notice that both invocations of the operation store() are issued by p_i, which is a sequential process). $view_1 < view_2$ if $view_1 \leq view_2$ and $view_1 \neq view_2$.

The store() **and** collect() **operations: definition** A store-collect object is formally defined by the following three properties, where the notations such as $inv[\text{st}] <_H resp[\text{col}]$ refer to the order on events as defined in Chap. 4:

- Validity. Let col be an invocation of collect() that returns the set $view$. For any $(i, v) \in view$, there is an invocation st of the operation store() with actual parameter v that was issued by the process p_i and this invocation has started before the invocation col terminates (i.e., $inv[\text{st}] <_H resp[\text{col}]$).

  This property means that a collect() operation can neither read from the future nor output values that have not yet been deposited.

- Partial order consistency. Let col_1 and col_2 be two invocations of the collect() operation that return the views $view_1$ and $view_2$, respectively. If col_1 terminates before col_2 starts (i.e., $resp[\text{col}_1] <_H inv[\text{col}_2]$), then $view_1 \leq view_2$.

  This property expresses the mutual consistency of non-concurrent invocations of the operation collect(): an invocation of collect() cannot obtain values older than the values obtained by a previous invocation of collect(). On the contrary, there is no constraint on the views returned by concurrent invocations of collect() (hence the name "partial order" for this consistency property).

- Freshness. Let st and col be invocations of the operations store(v) and collect() issued by p_i and p_j, respectively, such that st has terminated before col has started (i.e., $resp[\text{st}] <_H inv[\text{col}]$). The view returned by p_j contains a pair (i, v') such that v' is v or a value deposited by p_i after v.

  This property expresses the fact that the views returned by the invocations of the operation collect() are up to date in the sense that, as soon as a value was deposited, it cannot be ignored by future invocations of collect(). If store(v) is executed by a process p_i, the pair (i, v) must appear in a returned view (provided there are enough invocations of collect()) unless v was overwritten by a more recent invocation of store() issued by p_i.

- Liveness. Any invocation of an operation by a process that does not crash terminates.

```
operation store($v$) is
    $REG[i] \leftarrow v$; return()
end operation.

operation collect() is
    $view \leftarrow \emptyset$;
    for each $j \in \{1, \dots, n\}$ do
       if $(REG[j] \neq \perp)$ then $view \leftarrow view \cup \{(j, REG[j])\}$ end if
    end for;
    return($view$)
end operation.
```

Fig. 7.6 A trivial implementation of a store-collect object (code for p_i)

A trivial implementation A store-collect object can be trivially built from SWMR atomic read/write registers. This implementation is based on an array *REG*[1..*n*] with one entry per process. Only p_i can write *REG*[*i*], while all the processes can read it. Each entry is initialized to a default value $\perp$ (which cannot be deposited by a process). The construction is described in Fig. 7.6. It is self-explanatory. To collect values, a process reads the entries of the array *REG*[1..*n*] in any order.

A store-collect object has no sequential specification Differently from objects that are atomic, a store-collect object has no sequential specification. This means that it is not possible to express all the correct behaviors of a store-collect object using sequences on its operations (traces). This comes from the fact that the views returned by the collect operations are required to satisfy a partial order and not a total order. It follows that store-collect objects are not atomic objects.

To illustrate the fact that a store-collect object cannot have a sequential specification, let us consider the execution described in Fig. 7.7. There are five processes, p_1 to p_5. The invocations of the store() and collect() operations are explicitly indicated. The process p_5 never deposits a value. A "read line" (dotted line) is associated with each invocation of collect(). Its meaning is the following: the value obtained from a

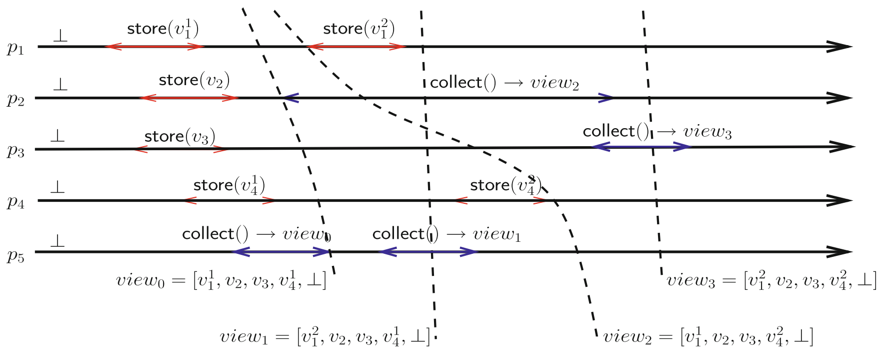


Fig. 7.7 A store-collect object has no sequential specification

process p_i is the last value stored by p_i before p_i's axis is cut by the corresponding read line. For convenience, a view is represented by an array ($\perp$ means that the process associated with the corresponding entry has never deposited a value). The value returned by an invocation of collect() is indicated after the arrow following the corresponding invocation.

It is easy to see that we have $view_0 \leq view_1 \leq view_3$, from which we can conclude that the invocations of collect() by p_5 and p_3 and the invocations of store() from which they obtained their views can be totally ordered. Similarly, as $view_0 \leq view_2 \leq view_3$, the first invocation of collect() by p_5 and the invocation of collect() by p_2 and p_3, plus invocations of store() from which they obtained their views can also be totally ordered.

Differently, we have neither $view_1 \leq view_2$ nor $view_2 \leq view_1$. These views are not ordered. While both $view_1$ and $view_2$ are correct views, ordering one before the other would make the other incorrect. It follows that the corresponding invocations of collect() cannot be ordered, preventing a sequential specification of a store-collect object.

7.2.2 An Adaptive Store-Collect Implementation

The adaptive implementation described in this section is due to H. Attiya, A. Fouren, and E. Gafni (2002).

Notion of an adaptive algorithm Considering the previous construction based on atomic read/write registers (Fig. 7.6), we can measure the cost of its store() and collect() operations by counting the number of read and write accesses to the base atomic registers. We then have the following: the cost of store() is constant (one write of a single base register), while the cost of collect() is linear with respect to the total number of processes, namely $O(n)$, as each entry of the array has to be read once. This means that, while the cost of a store is independent of n, the cost of a collect is proportional to n, whatever the number of processes that have stored values.

When looking at applications that use store-collect objects, the values that are deposited in a store-collect object can be arbitrary large, and what we abstract here as an atomic register $REG[i]$ is usually a block of a disk shared by the processes. As disk accesses are expensive, an efficient implementation of a disk-based store-collect object would be one such that the cost of the operation collect() would depend on the number of processes that have stored values, and not on the total number of processes.

This observation motivates the design of *adaptive* implementations. Those are such that, if some processes never access the object, the cost of an operation does not depend on them. More precisely, let k be the number of processes that have deposited values. An implementation of a store-collect object is adaptive if there is a function $f()$ such that the costs of the operations store() and collect() are upper-bounded by $O(f(k))$. This section presents an adaptive implementation of a store-collect object, based on atomic read/write registers, where both the cost of any invocation of the

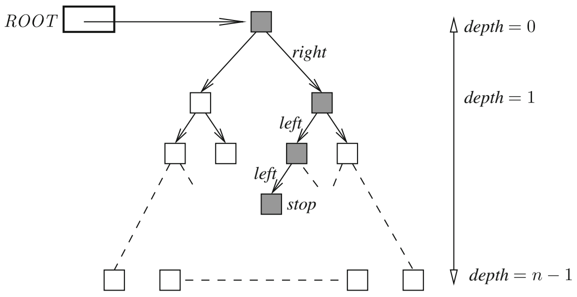


Fig. 7.8 A complete binary tree to implement a store-collect object

collect() operation and, for each process, the cost of its first store() operation are $O(k)$, while the cost of any other store() operation is $O(1)$.

Internal representation of a store-collect object This representation is based on a complete binary tree of depth $n - 1$ (Fig. 7.8). Only a part of the tree is used in a given execution, but this part is not known in advance, it is dynamically defined according to the pattern of the accesses issued by the processes (the black vertices in Fig. 7.8 denote the vertices which are currently used). The read-only shared register *ROOT* contains a pointer to the root of the tree.

The idea is that the first invocation of the operation store() by a process is a simple descent in the tree (without ever backtracking) from the root until a node (also called a vertex in the following) where (due to some predicate) the descent stops. An invocation of the operation collect() is a traversal of the part of the tree defined by the vertices already visited by invocations of the operation store(). Each vertex *VTX* is made up of the following fields (Fig. 7.9):

- *VTX.marked* is a Boolean atomic register (initialized to *false*) whose value indicates whether or not the vertex has been attributed to a process to deposit its successive values. Once a vertex was attributed to a process, it is assigned to that process forever, and consequently only that process can deposit a value in that

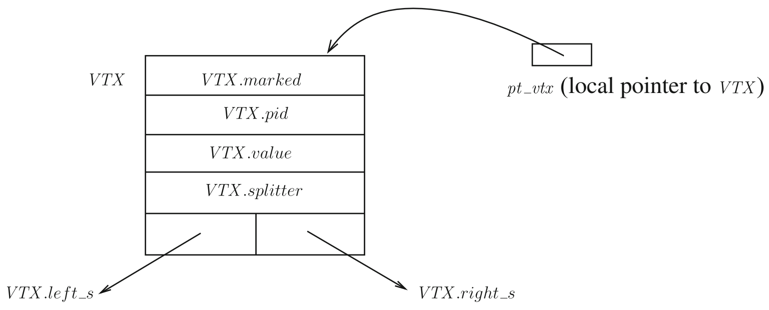


Fig. 7.9 Structure of a vertex of the binary tree

vertex. A process is assigned a vertex when it invokes for the first time the operation store().

- $VTX.pid$ is a write-once atomic register destined to contain the identity of the process to which the vertex VTX was attributed.
- $VTX.value$ is an atomic register where the process to which the vertex VTX was attributed (i.e., p_{pid}) deposits its value each time it executes a store() operation. Initially, $VTX.value = \perp$, for any vertex VTX.
- $VTX.left\_s$ and $VTX.right\_s$ are atomic registers containing pointers to the left and the right successor vertices of VTX. They contain $\perp$, if VTX is a leaf.
- $VTX.splitter$ is a splitter object associated with the vertex VTX. This object (which can be built wait-free from read/write registers only) was introduced in Sect. 5.2.1. It is used to to direct a process p_i (when it executes its first store() operation) to a free vertex. The vertex assigned to p_i is the vertex whose splitter returns the value $stop$ to p_i. The values $left$ or $right$ returned by a splitter are the routing information which govern p_i's descent of the tree.

Let VTX be a vertex of the tree and pt_vtx a local pointer to it used by some process p_i. The notation $(pt\_vtx \downarrow).field$ is used instead of $VTX.field$, where $field$ is any field of VTX.

The algorithm implementing the operation store() The algorithm implementing the store() operation is described in Fig. 7.10. Each process p_i is provided with a local variable my_vertex_i (initialized to $\perp$) whose scope is the entire execution and whose aim is to contain the pointer to the vertex of the tree assigned to it (this means that p_i will always deposit its values at the same place, namely in the register $my\_vertex_i.value$). Each execution of a store() operation uses two local variables denoted pt_vtx (a pointer to a vertex of the tree) and dir, whose values are meaningful only during the execution of the corresponding invocation of the operation store().

The first time it invokes store(), p_i executes lines 2–11. Starting from the root, p_i descends along a path of the tree according to the routing information provided by the splitters associated with the vertices it traverses. It stops at the first vertex whose splitter returns the value $stop$. Due to the splitter property, p_i is then the only process thats stops at that vertex (line 7). It then assigns this vertex to its variable my_vertex_i (line 10) and indicates that it was attributed this vertex (line 11) (so the other processes are aware that the value deposited in $my\_vertex_i.value$ is from p_i). Finally, p_i deposits its value v in the register $my\_vertex_i.value$ (line 13).

If it has already been attributed a vertex of the tree when it invokes a store() operation (we have then $my\_vertex_i \neq \perp$ at line 1), p_i directly deposit its value v in the register $my\_vertex_i.value$. It is easy to see that, in that case, the cost of an execution of store() is constant.

Let us observe that, given a vertex VTX, the atomic registers $VTX.pid$ and $VTX.value$ are SWMR registers. Differently from the SWMR atomic registers which have been previously used where the single writer is statically defined, here the single writer is dynamically determined. The fact that no process is a priori aware of which registers are assigned to which processes is a price to be paid to obtain an adaptive implementation.

The algorithm implementing the operation collect() The algorithm implementing the collect() operation is described in Fig. 7.10. It is a simple depth-first search algorithm (line 15) that, starting at the root of the tree, traverses all the marked vertices while collecting the values deposited at these vertices.

Let us notice that it is possible that, while a vertex *VTX* has been marked and attributed to a process p_k, p_k crashes before depositing a value in the register *VTX*.$value$ or (due to asynchrony) takes a very long time before depositing a value

```
operation store($v$) is
(1)  if ($my\_vertex_i = \bot$) then
(2)     $pt\_vtx \leftarrow ROOT$;
(3)     repeat $(pt\_vtx \downarrow).marked \leftarrow true$;
(4)             $dir \leftarrow \big((pt\_vtx \downarrow).splitter\big)$.direction($i$);
(5)             case ($dir = left$)   then $pt\_vtx \leftarrow (pt\_vtx \downarrow).left\_s$
(6)                  ($dir = right$) then $pt\_vtx \leftarrow (pt\_vtx \downarrow).right\_s$
(7)                  ($dir = stop$)  then exit the repeat loop
(8)             end case
(9)     end repeat;
(10)    $my\_vertex_i \leftarrow pt\_vtx$;
(11)    $my\_vertex_i.pid \leftarrow i$
(12) end if;
(13) $my\_vertex_i.value \leftarrow v$;
(14) return()
end operation.

operation collect() is
(15) let $view$ = df_search($ROOT$); return($view$)
end operation.

internal operation df_search($pt\_vtx$) is % returns a view %
(16) if ($(pt\_vtx \downarrow).marked$) then
(17)    if ($(pt\_vtx \downarrow).value \neq \bot$)
          then $view_i \leftarrow \{\big((pt\_vtx \downarrow).pid, (pt\_vtx \downarrow).value\big)\}$ else $view_i \leftarrow \emptyset$  end if;
(18)    if $\big((pt\_vtx \downarrow).right\_s \neq \bot\big)$
          then $view_{right} \leftarrow$ df_search$\big((pt\_vtx \downarrow).right\big)$ end if;
(19)    if $\big((pt\_vtx \downarrow).left\_s \neq \bot\big)$
          then $view_{left} \leftarrow$ df_search$\big((pt\_vtx \downarrow).left\big)$ end if;
(20)    $view_i \leftarrow view_i \cup view_{left} \cup view_{right}$;
(21)    return($view_i$)
(22) else return($\emptyset$)
(23) end if
end operation.
```

Fig. 7.10 An adaptive implementation of a store-collect object (code for p_i)

in *VTX*.$value$. Let us also notice that it is possible that, after *VTX* was marked, the vertices *VTX*.*left_s* *VTX*.*right_s* have been attributed to other processes, independently of whether a value has or has not been deposited in *VTX*.$value$. This means that, even when *VTX*..$marked$ $\wedge$ *VTX*.$value = \perp$, the tree traversal has to progress (lines 18–19) along both descendants of *VTX* (possibly until a leaf according to which vertices are marked). It stops and backtracks when it encounters a non-marked vertex (i.e., a vertex not yet attributed to a process).

7.2.3 Proof and Cost of the Adaptive Implementation

This section proves that the previous implementation is correct and adaptive.

Theorem 28 *The implementation described in Fig. 7.10 is a correct wait-free implementation of a store-collect object.*

Proof Let us first consider the liveness property. The first time a correct process (i.e., a process which does not crash) invokes store() it executes the repeat loop (lines 3–9). The fact that is exits the loop follows from the two following observations: (a) there are at most n processes which access the store-collect object, (b) the height of the binary tree is n, and (c) the property of the splitters attached to each vertex which ensures that, if x processes access a splitter, at most $x - 1$ of them are going left, and at most $x - 1$ of them are going right. Hence, the first invocation of store() by a correct process always terminates. Moreover, as $my\_vertex_i \neq \perp$ after the first invocation by a process, its other invocations trivially terminate.

The proof that any invocation of collect() by a correct process terminates follows from the observation that the tree is bounded and the fields *left_s* and *right_s* of any if its leaves remain always equal to $\perp$, which stops the depth-first search at line 18 or 19, if not yet done before at another vertex.

The fact that a view cannot contain values read from the future follows directly from the text of the algorithms. The rest of the validity property (a value which is returned was deposited), the freshness property, and the partial order consistency property follow from the following observations:

- If the view $view$ returned by a collect() operation contains the pair (i, v), that pair was obtained in the **then** part of line 17 during the tree traversal. As shown by line 17 of the collect() operation, the corresponding vertex was previously marked.
- A vertex *VTX* is marked during invocations of the operation store() (line 3). Moreover, as no process resets *VTX*.$marked$ to *false*, once marked, a vertex remains marked forever. It also keeps forever in *VTX*.pid (line 11) the identity i of the only process that obtains the value $stop$ from the associated splitter (line 7). That vertex is then definitely attributed to p_i (line 10) for it to deposit its future values in *VTX*.$value$ (line 13).

- When a process p_i stores a new value in the vertex assigned to it (i.e., the vertex *VTX* such that $VTX.pid = i$ and pointed to by my_vertex_i), it overwrites the previous value (if any).
- If a store() operation issued by p_j terminates before a collect() operation starts, the tree traversal generated by that collect() visits the vertex attributed to p_j (this follows from the fact that any invocation of collect() visits all the vertices that are currently marked).

It follows from the previous items that, if $view$ is returned by an invocation of collect() and contains the pair (i, v), (a) the value v was written by p_i in the vertex *VTX* such that $VTX.pid = i$ and (b) v the last value written by p_i before the invocation of collect() reads $VTX.value$. □

Notation Let k denote the number of processes that invoke the operation store() in a run.

Lemma 9 *If the depth of a vertex VTX is d,* $0 \leq d \leq k$, *then at most* $k - d$ *processes access the vertex VTX among the k processes that invoke the* store() *operation.*

Proof The proof is by induction on d, the depth of the vertex *VTX*. The base case $d = 0$ follows from the fact that k processes invoke the operation store() (no more than k processes access the root).

Assuming that the lemma holds for the vertices at depth d, $0 \leq d < k$, let us consider a vertex *VTX* at depth $d + 1 \leq k$. Let VTX' be the vertex parent of *VTX* in the tree. The depth of VTX' is d, and due to the induction assumption, at most $k - d$ processes access VTX'. Letting *VTX* be the left or right successor of VTX' in the tree, it follows from the property of the splitter associated with VTX' that at most $k - d - 1 = k - (d + 1)$ processes access the vertex *VTX*, which proves the induction case. □

Lemma 10 *If a process writes its identity in a vertex (line 13), the depth of that vertex is smaller than k, and no other process writes its identity in the same vertex.*

Proof Let us first notice that a process p_i writes its identity in a vertex only when it invokes the store() operation for the first time. It follows from Lemma 9 (taking $d = k$) that no process accesses a vertex at depth k. Hence, p_i stops at a vertex with depth smaller than k. Moreover, due to the splitter property, at most one process stops at a given vertex. It follows that no two processes are assigned the same vertex. □

Lemma 11 *All the vertices from the root to a marked vertex are marked.*

Proof If a vertex *VTX* is marked, some process p_k has set $VTX.marked$ to *true*. It follows from line 3 inside the **repeat** loop of the operation store() that p_k has also marked all the vertices from the root until *VTX*. The lemma follows from this observation and the fact that no process ever resets a marked vertex to *false*. □

Theorem 29 *Let* p_i *be a process that invokes the* store() *operation. The cost of its first invocation is* $O(k)$. *The cost of its following invocations is* $O(1)$.

Proof Let us consider the first invocation of the store() operation issued by a process p_i. It descends from the root until the first vertex *VTX* such that *VTX*.$marked$ is equal to *false* (due to Lemma 9, such a non-marked vertex is visited by p_i; this vertex is at depth $k-1$ in the "worst" case). As all the vertices (but the last one) visited by p are marked (Lemma 11), and there are at most $k-1$ processes that have invoked the store() operation before p_i, it follows from Lemma 10 that p_i executes at most $k-1$ iterations of the **repeat** loop (lines 3–9). As both the statements executed $k-1$ times inside the loop and the statements executed once outside the loop are made up of a constant number of accesses to read/write registers, it follows that the total number of read/write operations on base atomic registers is upper bounded by $O(k)$. (Let us recall that the splitter operation direction() requires a bounded number of read/write accesses on the two atomic registers from which it is built.)

As soon as vertex was attributed to p_i, we have $my\_vertex_i \neq \perp$, and the cost of its following invocations of store() is constant, i.e., $O(1)$. □

Theorem 30 *The cost of an invocation of the operation* collect() *is* $O(k)$.

Proof Let us first observe that, for each vertex visited by an invocation of collect(), the number of operations on base atomic read/write registers is constant. So, to determine an upper bound on the number of operations on base atomic registers, we only need to compute an upper bound on the number of marked vertices visited by an invocation of collect(), when at most k processes have invoked the store() operation. (Let us observe that a vertex child of a marked vertex is visited even if it is not marked. As this can happens at most k times, this does not change the magnitude order of the cost of an invocation of the collect() operation.)

Thanks to the property of the splitter objects associated with each vertex and the fact that the vertices that are marked (by the store() operation) define a subtree rooted at *ROOT* (Lemma 11), an upper bound u_k on the number of marked vertices accessed by an invocation of the collect() operation can be defined as follows:

$$u_k = 1 + u_\ell + u_r \quad \text{with} \quad 0 \leq \ell, r \leq k-1 \quad \text{and} \quad \ell + r \leq k,$$

where (see Fig. 7.11 where the marked vertices are the black vertices):

- The number 1 comes from the fact that, as $k \geq 1$, the root is always marked. (Notice that it is possible that no process stops at the root.)
- u_ℓ (or u_r) is the number of marked vertices in the left (or right) subtree of the root.

It is easy to see that $u_0 = 0$, $u_1 = 1$, and $u_2 = 3$.

Let us consider the more constraining recurrence equation

$$u_k = a + u_\ell + u_r \quad \text{with} \quad 1 \leq \ell, r \leq k-1 \quad \text{and} \quad \ell + r = k.$$

It is easy to verify that $u_k = (k-1)\,a + k\,u_1$ is the unique solution. More precisely, it is trivial to see that it is verified for $k = 1$. Then, by strong induction, developing u_ℓ and u_r we obtain

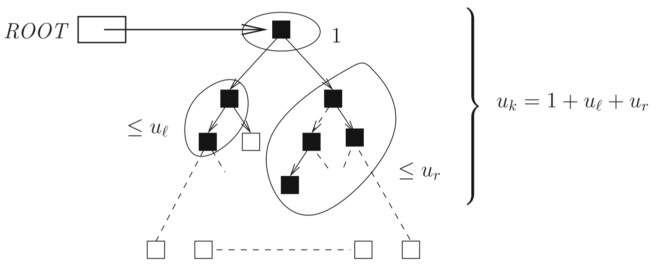


Fig. 7.11 Computing an upper bound on the number of marked vertices

$$\begin{aligned} u_k &= a + u_\ell + u_r \\ &= a + \big((\ell - 1)a + \ell\, u_1\big) + \big((r-1)a + r\, u_1\big) \\ &= (\ell + r - 1)a + (\ell + r)u_1 \\ &= (k-1)\, a + k\, u_1. \end{aligned}$$

Replacing a and u_1 by their values, namely 1, we obtain $u_k = 2k - 1$. It follows that $u_k = 2k - 1$ is an upper bound on the number of marked vertices that are accessed by an invocation of the collect() operation. The number of accesses to base atomic read/write registers is consequently upper-bounded by $O(k)$. $\square$

7.3 Fast Store-Collect Object

This section presents the notion of a *fast store-collect* object and an efficient wait-free implementation of it.

7.3.1 Fast Store-Collect Object: Definition

Merging the operations store() **and** collect() In some applications, each time a process deposits a new value, it also needs to collect the last values that have been deposited. More specifically, the processes are interested in using a single operation, denoted store_collect(). As shown in Fig. 7.12, such an operation can be easily built from an array of SWMR atomic read/write registers by a simple merge of the algorithms described in Fig. 7.6.

Looking for efficiency when there is no concurrency As we can see, each invocation of the store_collect() operation costs $O(n)$ read/write accesses to base atomic registers. It appears that, in some applications, there are periods during which a single process invokes store_collect() (that process possibly playing a special role,

```
operation store_collect($v$) is
    $REG[i] \leftarrow v$;
    $view \leftarrow \{(i,v)\}$;
    for each $j \in \{1,\ldots,n\}, j \neq i$ do
        if $(REG[j] \neq \perp)$ then $view \leftarrow view \cup \{(j, REG[j])\}$ end if
    end for;
    return($view$)
end operation.
```

Fig. 7.12 Merging store() and collect() (code for process p_i)

e.g., leader, during the corresponding period). Such periods are characterized by a *concurrency degree* (i.e., the number of processes invoking operations) equal to 1.

We are interested here in an implementation of the store_collect() operation whose cost eventually becomes constant when there is a single process that invokes it during a long enough period. This search for such an efficient implementation is motivated by the practical cases where the atomic registers $REG[1..n]$ are actually abstracting n physical blocks of one or several disks. It becomes evident that, in such a setting, a store_collect() operation whose cost is $O(1)$ read/write operations on disk blocks in concurrency-free scenarios is much more interesting than an implementation of store_collect() whose cost is $O(n)$ disk accesses.

7.3.2 *A Fast Algorithm for the* store_collect() *Operation*

The design of a fast store_collect() algorithm is based on two ideas. The first consists in adding a shared control variable and the second in considering a two-step algorithm. We present them in an incremental way. This algorithm is due to B. Englert and E. Gafni (2002).

Each process p_i maintains also a local array $reg_i[1..n]$ where it keeps a copy of the values that it considers as the last deposited values. This array acts as a local cache that can save accesses to shared atomic registers.

To simplify the text of the algorithm, this section considers that the view returned by store_collect() is an array of n values. It is trivial to obtain the corresponding (process identity, value) pairs by eliminating the pairs whose value is $\perp$.

First step: adding a MWMR atomic register The first step of the construction consists in adding a MWMR atomic register *LAST* whose aim is to contain the identity of the last process that has terminated executing store_collect() (initially, $LAST = \perp$). This will allow a process to learn if, since its previous invocation, another process has terminated execution of a store_collect() invocation. (When the array of atomic registers $REG[1..n]$ abstracts blocks of a shared disk, the atomic register *LAST* can be placed on the same disk.)

The underlying idea to obtain an efficient algorithm is the following. When it invokes store_collect(), a process p_i returns the last cached values if it is the last process that has previously terminated a store_collect() invocation (in that case, $LAST = i$). If this is not the case ($LAST \neq i$), at least one process has written a new value, and consequently p_i reads the last deposited values from the atomic registers.

```
$REG[i] \leftarrow v$; $reg_i[i] \leftarrow v$;
**if** ($LAST \neq i$) **then for each** $j \in \{1, \ldots, n\}, j \neq i$ **do** $reg_i[j] \leftarrow REG[j]$ **end for**;
                        $LAST \leftarrow i$
**end if**;
return($reg_i[1..n]$).
```

Unfortunately, this simple *LAST*-based algorithm is incorrect. This is easily shown in Fig. 7.13, where (as in Fig. 7.7) a read (dotted) line obtains the last value deposited by a process at the time this dotted line cuts the corresponding process axis. The figure considers that each p_i has previously deposited the value v_i^0. So, as a view contains a value per process, it is represented by an array. (The values of the variable $step_4$ and the views $view_3$ and $view_4$ are meaningless for the present discussion, they will be used later.)

Let $view_1 = [v_1^0, v_2^0, v_3^0, v_4^1]$ be the view returned to p_4 by its invocation store_collect(v_4^1), $view_2$ be the view returned to p_1 by its invocation store_collect (v_1^1), and $view'_3$ and $view'_4$ be the views returned to p_4 by its last two consecutive invocations store_collect(v_4^2) and store_collect(v_4^3), respectively.

In this execution, p_1 obtains $view_2$ after p_4 has computed $view_1$ but before it has updated *LAST* to 4: p_4 paused after having read asynchronously one after the other the registers $REG[1..n]$ and before updating *LAST* and the store_collect() issued by p_1 occurred during this pause.

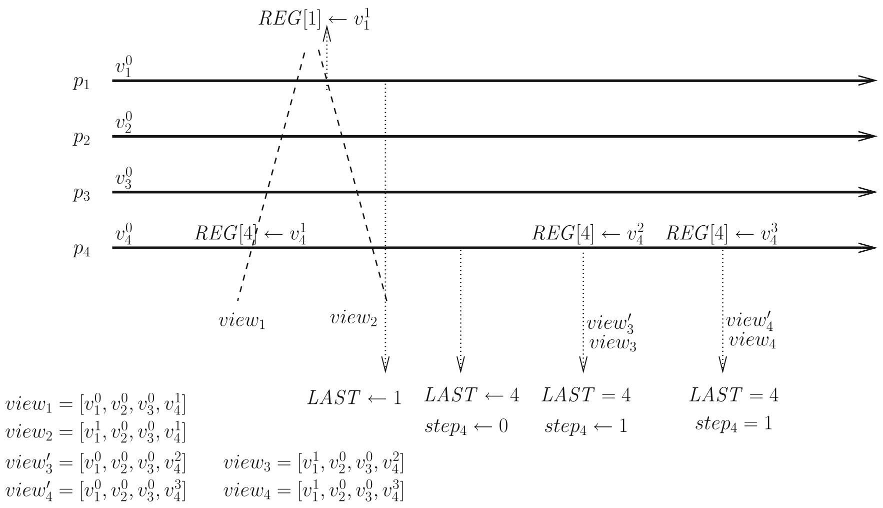

$REG[1] \leftarrow v_1^1$
p_1
v_1^0
p_2
v_2^0
p_3
v_3^0
p_4
v_4^0
$REG[4] \leftarrow v_4^1$
$REG[4] \leftarrow v_4^2$
$REG[4] \leftarrow v_4^3$
$view_1$
$view_2$
$view'_3$
$view_3$
$view'_4$
$view_4$
$LAST \leftarrow 1$
$LAST \leftarrow 4$
$step_4 \leftarrow 0$
$LAST = 4$
$step_4 \leftarrow 1$
$LAST = 4$
$step_4 = 1$
$view_1 = [v_1^0, v_2^0, v_3^0, v_4^1]$
$view_2 = [v_1^1, v_2^0, v_3^0, v_4^1]$
$view'_3 = [v_1^0, v_2^0, v_3^0, v_4^2]$
$view'_4 = [v_1^0, v_2^0, v_3^0, v_4^3]$
$view_3 = [v_1^1, v_2^0, v_3^0, v_4^2]$
$view_4 = [v_1^1, v_2^0, v_3^0, v_4^3]$

Fig. 7.13 Incorrect versus correct implementation of the store_collect() operation

It is easy to see that $LAST = 4$ when the invocation that returns $view'_3$ starts. Consequently, that invocation returns the values cached at p_4, kept in $reg_4[1..n]$, missing the value v_1^1 deposited by the invocation of store_collect() issued by p_1. As that invocation by p_1 is terminated before the invocation of store_collect() issued by p_4 (which returns $view'_3$) started, it follows that the store_collect() returning $view'_3$ is incorrect as it violates the freshness property (it does not return the last values deposited by the invocations of store_collect() that have terminated before it started). Similarly, the view $view'_4$ returned by the next invocation of store_collect() issued by p_4 is incorrect.

Second step: a two-step algorithm The previous problem arises because, on one side, processes can overwrite each other when writing the MWMR atomic register $LAST$ and, on another side, the reading of the whole array $REG[1..n]$ is not an atomic operation. A way to solve this problem consists in forcing a process p_i to access the array of atomic registers $REG[1..n]$ even when $LAST = i$, in the cases where $LAST \neq i$ when p_i issued its previous invocation of store_collect(). This means that a process p_i can safely read the values from its local cache $reg_i[1..n]$ only when it sees it is the last writer ($LAST = i$) twice in a row. This means that during the first of these two store_collect(), p_i has obtained and saved in its local cache the last values that have been deposited.

The resulting algorithm for the store_collect() operation is described in Fig. 7.14. Each process p_i maintains a local variable $step_i$ whose scope is the whole execution. Its aim is to track whether p_i remains the last writer between its successive invocations of store_collect(). Initialized to 0, this local variable is set to 1 when $LAST = i \wedge step_i = 0$ (i.e., when p_i considers it is the last writer) while it was not the last writer before. Then, when $LAST = i \wedge step_i = 1$, p_i considers that it is still the last writer since its previous invocation of store_collect(). As we have seen, this is the case where p_i can read from its local cache.

As an example of the way the algorithm works, let us again consider Fig. 7.13 and observe the management of the local variable $step_4$. One can see that p_4 has to read the array of atomic registers $REG[1..n]$ before returning $view_3$. This is because we have then $LAST = 4 \wedge step_4 = 0$. Differently, it does not have to read that array

```
operation store_collect($v$) is
    $REG[i] \leftarrow v$; $reg[i] \leftarrow v$;
    case ($LAST \neq i$) then for each $j \in \{1,\ldots,n\}, j \neq i$ do $reg_i[j] \leftarrow REG[j]$ end for;
                             $LAST \leftarrow i$; $step_i \leftarrow 0$
         ($LAST = i$) $\wedge$ ($step_i = 0$) then
                             for each $j \in \{1,\ldots,n\}, j \neq i$ do $reg_i[j] \leftarrow REG[j]$ end for;
                             $step_i \leftarrow 1$
         ($LAST = i$) $\wedge$ ($step_i = 1$) then no-op
    end case;
    return($reg_i[1..n]$)
end operation.
```

Fig. 7.14 An efficient store_collect() algorithm (code for p_i)

during its next invocation of store_collect(). As we then have $LAST = 4 \wedge step_4 = 1$, the correct value $view_4$ is obtained from the local cache.

7.3.3 Proof of the Fast Store-Collect Algorithm

Lemma 12 *The implementation described in Fig. 7.14 satisfies the freshness property of a store-collect object.*

Proof Let $view_i$ be the vector view returned by an invocation $\mathsf{st\_col}_i^1$ of store_collect() issued by a process p_i. Let store_collect(v_j) be the last invocation by p_j that terminated before $\mathsf{st\_col}_i^1$ started. We have to show that $view_i[j]$ contains v_j or a value v'_j such that p_j has invoked store_collect(v'_j) after store_collect(v_j).

The lemma is trivially satisfied if $\mathsf{st\_col}_i^1$ reads all the atomic registers $REG[1..n]$. So, let us consider the case where $\mathsf{st\_col}_i^1$ is such that $LAST = i \ \wedge \ step_i = 1$. Let $\mathsf{st\_col}_i^0$ be the invocation of store_collect() by p_i that precedes $\mathsf{st\_col}_i^1$ and is such that p_i reads $LAST \neq i$. Such an invocation exists because $LAST$ is initialized to $\perp$. Moreover, there is another invocation $\mathsf{st\_col}_i^2$ issued by p_i that is in between $\mathsf{st\_col}_i^0$ and $\mathsf{st\_col}_i^1$ (this follows from the management of $step_i$), and p_i read all the atomic registers $REG[1..n]$ when it executed $\mathsf{st\_col}_i^2$. As any process p_j that invokes store_collect() first deposits its value in $REG[j]$ and only then writes its identity j in $LAST$, it follows that the last invocation of store_collect() issued by p_j which terminates before $\mathsf{st\_col}_i^1$ started has terminated before $\mathsf{st\_col}_i^0$. Consequently, the value read from $REG[j]$ by $\mathsf{st\_col}_i^1$ is v_j or a more recent value. $\square$

Lemma 13 *The implementation described in Fig. 7.14 satisfies the partial order property of a store-collect object.*

Proof Let st_col and $\mathsf{st\_col}_i^2$ be two invocations of store_collect() such that st_col terminates before $\mathsf{st\_col}_i^2$ starts. Moreover let p_i be the process that has invoked $\mathsf{st\_col}_i^2$ (hence the index i). We have to show that, for every process p_j, the value returned by $\mathsf{st\_col}_i^2$ is not older than the value returned by st_col.

If $\mathsf{st\_col}_i^2$ returns values just read from the atomic registers $REG[1..n]$, the lemma immediately follows. So, let us consider the case where $\mathsf{st\_col}_i^2$ does not read the atomic registers $REG[1..n]$ (see Fig. 7.15).

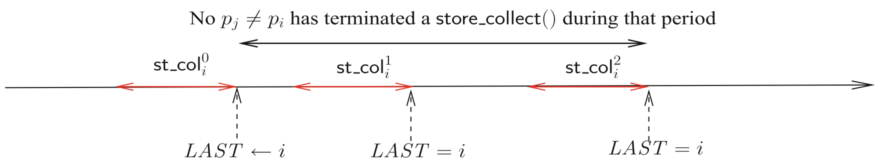


Fig. 7.15 Sequential and concurrent invocations of store_collect()

This means that p_i reads $LAST = i$ and $step_i = 1$ when it executes $\mathsf{st\_col}_i^2$. As in the previous lemma, let us consider the last invocation $\mathsf{st\_col}_i^0$ by p_i such that $LAST \neq i$. We have seen that $\mathsf{st\_col}_i^0$ does exist. As p_i has read all the atomic registers $REG[1..n]$ during $\mathsf{st\_col}_i^0$, its invocations after $\mathsf{st\_col}_i^0$ do not return values older than the ones returned by $\mathsf{st\_col}_i^0$. Hence, if st_col terminates before $\mathsf{st\_col}_i^0$ starts, the lemma follows. Consequently, we have to consider only the invocations st_col that are concurrent with $\mathsf{st\_col}_i^0$ or start after $\mathsf{st\_col}_i^0$ has terminated. Let us observe that these invocations do not terminate between the end of $\mathsf{st\_col}_i^0$ (where p_i writes its identity in $LAST$) and $\mathsf{st\_col}_i^2$ (where p_i finds $LAST = i$), otherwise p_i would not read $LAST = i$. It follows that these invocations are concurrent with $\mathsf{st\_col}_i^2$ and $\mathsf{st\_col}_i^1$ (the invocation issued by p_i in between $\mathsf{st\_col}_i^0$ and $\mathsf{st\_col}_i^2$ whose existence was proved in the previous lemma) or start after $\mathsf{st\_col}_i^1$ and are concurrent with $\mathsf{st\_col}_i^2$. It follows that all the invocations st_col that have terminated before $\mathsf{st\_col}_i^2$ starts have terminated before $\mathsf{st\_col}_i^0$ terminates and updates $LAST$ to i. As $\mathsf{st\_col}_i^2$ returns the values read by $\mathsf{st\_col}_i^1$ from the atomic registers, it follows that this invocation cannot return values older than the ones returned by st_col, which concludes the proof of lemma. □

Remark As suggested in the previous lemma, the fact that a process reads from its local cache does not mean that there are no concurrent invocations of store_collect() while a process returns the values kept in its local cache. To visualize this, let us consider the execution depicted in Fig. 7.16. This execution is nearly the same as the one described in Fig. 7.13. The only difference is that there is now an additional invocation of store_collect() by process p_2 and this invocation lasts a long time and is concurrent with all the other invocations of store_collect(). This invocation by

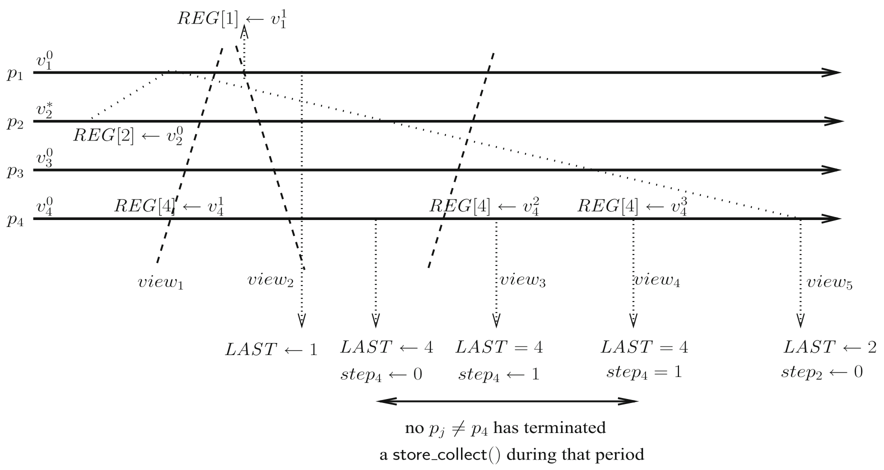


Fig. 7.16 Concurrent invocations of store_collect()

p_2, which is represented by a bold dotted line, is the one that defines v_2^0 as the last value of p_2 (assuming now that its previous value was v_2^*). The view it returns is the vector $view_5 = [v_1^0, v_2^0, v_3^0, v_4^3]$ which is comparable to neither $view_3$ nor $view_4$ (the vectors returned by the two consecutive invocations of store_collect() issued by p_4).

Theorem 31 *The implementation described in Fig. 7.14 is a wait-free implementation of a store-collect object where the* store() *and* collect() *operations are merged into a single* store_collect() *operation.*

Proof The algorithm is trivially wait-free. The proof of the validity property is the similar to the validity proof of Theorem 28. The proofs of the partial order consistency and freshness properties follow from Lemmas 12 and 13. □

Theorem 32 *If there is a time τ_0 after which there is a single process p_i that invokes* store_collect()*, then there exists a time $\tau_1 > \tau_0$ after which the cost of each of its invocations is $O(1)$ accesses to atomic registers.*

Proof The proof is an immediate consequence of the management of the atomic register *LAST* and the local variable $step_i$. If, after some time, only p_i invokes store_collect() operations, *LAST* remains permanently equal to i. It then follows from the management of $step_i$ that, after two invocations of store_collect() by p_i, we always have $LAST = i \;\wedge\; step_i = 1$. From that time, each time it invokes store_collect(), p_i writes its new value into $REG[i]$ and reads *LAST* (and learns that it is always the last writer). There are then two accesses to atomic registers, whatever the value of n. □

7.4 Summary

This chapter has considered the base read/write asynchronous computation model in which any number of processes may crash. Hence, implementations cannot use strong base atomic operations such as compare&swap(), they can rely only on atomic read/write registers.

Wait-free implementations suited to such a weak computation model have been presented. The first considered a weak counter which can be accessed by infinitely many processes. The second considered store-collect objects and fast store-collect objects.

7.5 Bibliographic Notes

- The notion of a weak counter for infinitely many processes and the corresponding implementation are due to M.K. Aguilera [14].

- The notions of infinitely many processes and finite concurrency used in this chapter are due to E. Gafni, M. Merritt, and G. Taubenfeld [110, 204].
- The notion of an adaptive algorithm and its underlying theory is investigated in [30, 34].
- The adaptive store-collect object presented in this chapter is due to H. Attiya, A. Fouren, and E. Gafni [35].
- Long-lived adaptive store-collect objects are presented in [12].
- The fast implementation of the store_collect() operation is due to B. Englert and E. Gafni [94] who proposed it to improve the disk version of the Paxos algorithm [109].

7.6 Problem

1. Design and prove correct a read/write-based wait-free implementation of a multi-shot weak counter object that provides its users with the two operations increment() and fast_get_count().

Chapter 8
Snapshot Objects from Read/Write Registers Only

This chapter is devoted to *snapshot* objects. Such an object can be seen as a store-collect object whose two operations are atomic. After having defined the concept of a snapshot object, this chapter presents wait-free implementations of it, which are based on atomic read/write registers only. This chapter introduces also the notion of an *immediate snapshot object*, which can be seen as the atomic counterpart of the fast store-collect object presented in the previous chapter.

Keywords Atomic snapshot object · Immediate snapshot object · Infinitely many processes · One-shot object · Read/write system · Recursive algorithm

8.1 Snapshot Objects: Definition

Snapshot object A snapshot object is an object that consists of m components (each component being an atomic read/write register). It provides the processes with two operations denoted update() and snapshot(). The update() operation allows the invoking process to store a new value in a given component, while the snapshot() operation allows it to obtain the values of all the components as if that operation was executed instantaneously.

The invocations of the operations update() and snapshot() are atomic: to an external observer, they appear as if they executed one after the other, each being associated with a point of the time line lying between its start event and its end event (as defined in Chap. 4).

Hence, while store-collect objects are not atomic objects, snapshot objects are. They can consequently be defined by a sequential specification made up of all the traces of update() and snapshot() associated with correct behaviors (each invocation of snapshot() returns the last values of each component that were deposited before that invocation).

Single-writer snapshot object A single-writer snapshot object has one component per object, hence $m = n$, and while a process can read any component it can write

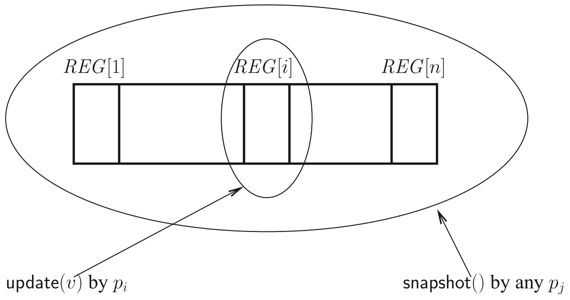


Fig. 8.1 Single-writer snapshot object for n processes

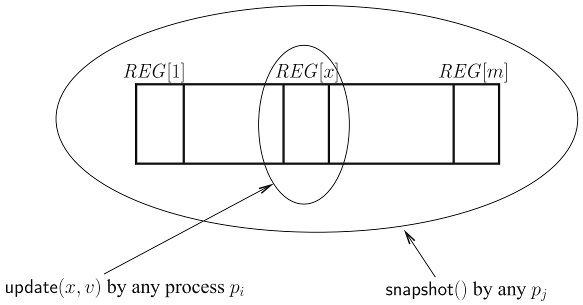


Fig. 8.2 Multi-writer snapshot object with m components

only the component associated with it. In this case, the base atomic read /write registers from which the snapshot object is built are SWMR registers.

As shown in Fig. 8.1, a process p_i invokes `update`(v) to assign a new value to the SRMW register $REG[i]$ and any process p_j invokes `snapshot`() to read atomically the whole array $REG[1..n]$.

Multi-writer snapshot object A multi-writer snapshot object is a snapshot object for which each component can be written by any process. It follows that the base atomic read /write registers from which a multi-writer snapshot object is built are MWMR registers.

A multi-writer snapshot object with m components is described in Fig. 8.2. A process p_i invokes `update`(x, v) to assign the value v to the MWMR component $REG[x]$ and invokes `snapshot`() to read atomically the whole array $REG[1..m]$.

8.2 Single-Writer Snapshot Object

Assuming a system of n processes, this section presents a wait-free implementation of a single-writer snapshot object due to Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit (1993). This implementation is presented in an incremental way.

8.2.1 An Obstruction-Free Implementation

Let $REG[1..n]$ be the array which is the internal representation of the snapshot object. A simple principle to be able to distinguish different updates of $REG[i]$ by process p_i consists in considering that each atomic register $REG[i]$ is made up of two fields, $REG[i].val$, which contains the last value written by p_i, and $REG[i].sn$, its associated sequence number.

The corresponding algorithm for the update() operation is described in Fig. 8.3. The local variable denoted sn_i, which is initialized to 0, allows p_i to generate sequence numbers.

The algorithm implementing the operation snapshot() is based on what is called a "sequential double collect" which is made up of two consecutive invocations of the function collect(). As we have seen in the previous section, this operation reads asynchronously all the registers of the array $REG[1..n]$ (lines 10–11). (Its trivial implementation considered in the implementations that follow can be replaced by a more efficient adaptive implementation as seen previously).

An invocation of snapshot() repeatedly reads twice the array $REG[1..n]$ (lines 4 and 6 or lines 8 and 6) until two consecutive collects obtain the same sequence number values for each register $REG[j]$ (the local arrays aa_i and bb_i are used to save the values read from the array in the first and second collect, respectively). When, $\forall j : aa_i[j] = bb_i[j]$, the corresponding double collect is said to be *successful* and the algorithm then returns the array of values $[aa_i[1].val, \ldots, aa_i[n].val]$ (line 7).

```
operation update($v$) is
(1)  $sn_i \leftarrow sn_i + 1$;
(2)  $REG[i] \leftarrow \langle v, sn_i \rangle$;
(3)  return()
end operation.

operation snapshot() is
(4)  $aa_i \leftarrow$ collect();
(5)  repeat forever
(6)      $bb_i \leftarrow$ collect();
(7)      if ($\forall j : aa_i[j] = bb_i[j]$) then return($aa_i[1..n].val$) end if;
(8)      $aa_i \leftarrow bb_i$
(9)  end repeat
end operation.

internal operation collect() is
(10) for each $j \in \{1, \ldots, n\}$ do $reg[j] \leftarrow REG[j]$ end for;
(11) return($reg$)
end operation.
```

Fig. 8.3 An obstruction-free implementation of a snapshot object (code for p_i)

Theorem 33 *The algorithms described in Fig. 8.3 define an obstruction-free implementation of an atomic snapshot object.*

Proof Let us first observe that an invocation of an update() operation issued by a correct process always terminate. Let us assume that, after some time, only processes that have invoked snapshot() issue computation steps. It follows that the sequence numbers are no longer modified and consequently any process that executes steps of the operation snapshot() eventually executes a successful double collect and terminates its invocation. It follows that the implementation is obstruction-free.

Let us now show that the implementation provides an atomic snapshot object. To that end, let us define the linearization points on the invocations of the update() and snapshot() operations as follows. As the base registers are atomic, we can consider that their read and write operation occur instantaneously at some point in time.

- Let the linearization point of an update() operation issued by p_j be the time instant when p_j writes $REG[j]$.
- Considering the invocation of a snapshot() operation that returns at line 7, let collect$_1$ and collect$_2$ be its last two invocations of collect(). Moreover, let τ_b^i (or τ_e^i) be the time at which collect$_1$ (or collect$_2$) read $REG[i]$. As both collect$_1$ and collect$_2$ obtain the same sequence number from $REG[i]$, we can conclude that, for any i, no update() operation issued by p_i has terminated and modified $REG[i]$ between τ_b^i and τ_e^i. As collect$_2$ starts after collect$_1$ has terminated, it follows that, between $\max(\{\tau_b^i\}_{1\leq i\leq n})$ and $\min(\{\tau_e^i\}_{1\leq i\leq n})$, no atomic register $REG[i]$, $1 \leq i \leq n$, was modified. It is consequently possible to associate with the corresponding invocation of the snapshot() operation a linearization point τ of the time line such that $\max(\{\tau_b^i\}_{1\leq i\leq n}) < \tau < \min(\tau_e^i\}_{1\leq i\leq n})$. Hence, from an external observer point of view, we can consider that the invocation of snapshot() occurred instantaneously at time τ (see Fig. 8.4) after all the invocations of the first update() operation and before the invocations of the second update() operation.

It follows from the previous definition of the linearization points that the operations that terminate define an atomic snapshot object. □

Remark As just noticed, an invocation of update() by a correct process always terminates. Differently, it is possible that invocations of the snapshot() operation never terminate. Let snap be such a non-terminating invocation. This can occur when one or several processes invoke continuouslyupdate() operations in such a way that the termination predicate $\forall j : aa_i[j].sn = bb_i[j].sn$ is never satisfied when checked by the snap. It is important to see that this is not due to the fact that processes crash, but to the fact that processes invoke continuously the update() operation. This is a typical starvation situation that has to be prevented in order to obtain a wait-free construction.

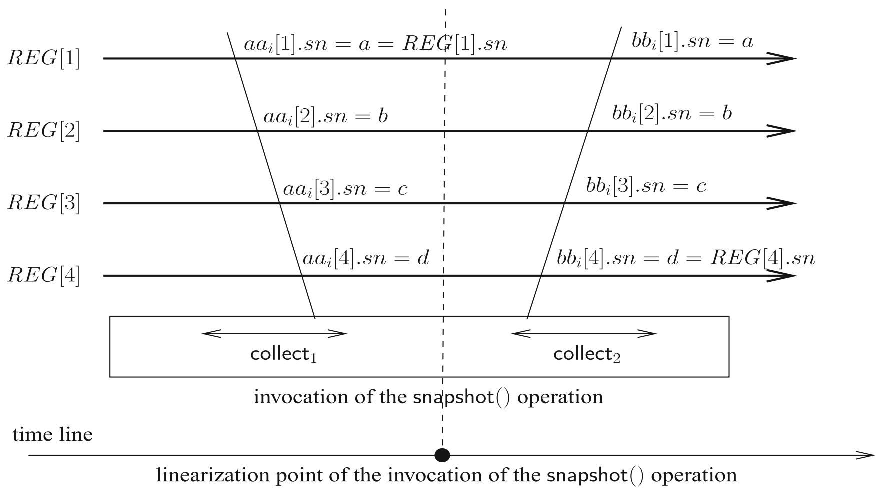

$REG[1]$
$REG[2]$
$REG[3]$
$REG[4]$
$aa_i[1].sn = a = REG[1].sn$
$aa_i[2].sn = b$
$aa_i[3].sn = c$
$aa_i[4].sn = d$
$bb_i[1].sn = a$
$bb_i[2].sn = b$
$bb_i[3].sn = c$
$bb_i[4].sn = d = REG[4].sn$
collect$_1$
collect$_2$
invocation of the snapshot() operation
time line
linearization point of the invocation of the snapshot() operation

Fig. 8.4 Linearization point of an invocation of the snapshot() operation (case 1)

8.2.2 *From Obstruction-Freedom to Bounded Wait-Freedom*

A basic principle to obtain a wait-free implementation (when possible) consists in using a helping mechanism (similarly to the one used in Sect. 7.1.3. Here, this "helping" principle can be translated as follows. If the are continuous invocations of the operation update() that could prevent invocations of the snapshot() operation from terminating, these invocations of update() have to help the invocations of snapshot() to terminate. The solution we present relies on this nice and simple idea.

The fact that a double collect is unsuccessful (i.e., does not allow an invocation of the snapshot() operation to terminate) can be attributed to invocations of the update() operation which increase sequence numbers, and consequently make false the test that controls the termination of the algorithm implementing the snapshot() operation.

Let us observe that, if an invocation of snapshot() (say snap) issued by a process p_i sees two distinct invocations of update() issued by the same process p_j (these updates upd$_1$ and upd$_2$ write distinct sequence numbers in $REG[j]$), we can conclude that upd$_2$ was entirely executed during the invocation snap (see Fig. 8.5). This is because the update by p_j of the base atomic register $REG[j]$ is the last operation executed in an update() operation.As the second update upd$_2$ is entirely executed during the execution of snap (it starts after it and terminates before it), the previous observations suggest requiring upd$_2$ to help snap. This help can be realized as follows:

- Each update is required to include an "internal" invocation of the snapshot() operation (see Fig. 8.5, where the rectangle inside the invocation upd$_2$ represents an internal invocation of the snapshot() operation –denoted int_snap– invoked by that update). Let *help* be the array of values obtained by the invocation int_snap.

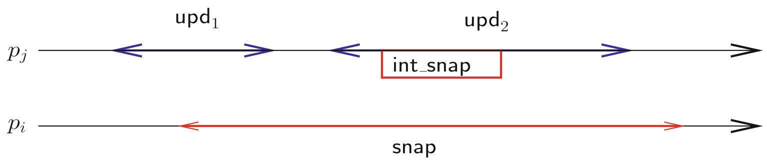


Fig. 8.5 The update() operation includes an invocation of the snapshot() operation

- As the invocation int_snap is entirely overlapped by the invocation snap, snap can borrow the array *help* and return it as its own result. (Notice it is possible that the internal snapshot invocation inside upd$_1$ may be also entirely overlapped by snap, but there is no way to know this.) This overlapping is important to satisfy the atomicity property; namely, the values returned have to be consistent with the real-time occurrence order of the operation invocations.

The proof will show that such an addition of an invocation of snapshot() to the algorithm implementing the update() operation is not at the price of the wait-free property.

The algorithms implementing the update() and snapshot() algorithms are described in Fig. 8.6. The function collect() is the same as before. A SWMR atomic register $REG[i]$ is now made up of three fields: $REG[i].val$ and $REG[i].sn$ as before, plus the new field $REG[i].help\_array$, whose aim is to contain a helping array as previously discussed. The final version of both update() and snapshot() algorithms is a straightforward extension of their previous attempt versions.

The main novelty lies in the local variable can_help_i that is used by a process p_i when it executes the operation snapshot(). The aim of this set, initialized to $\emptyset$, is to contain the identity of the processes that have terminated an invocation of update() since p_i started its current invocation of the snapshot() operation. Its use is described in the **if** statement at lines 12–17. More precisely, when a double collect is unsuccessful, p_i does the following with respect to each process p_j that made the double collect unsuccessful (i.e., such that $(aa_i[j].sn \neq bb_i[j].sn)$):

- If $j \notin can\_help_i$ (line 15). In this case, p_i discovers that p_j has terminated an invocation of update() since it started its invocation of the snapshot() operation. Consequently, if p_j terminates a new invocation of update() while p_i has not yet terminated its invocation of snapshot(), p_j can help p_i terminate.

- If $j \in can\_help_i$ (line 14). In this case, p_j has entirely executed an update() operation while p_i is executing its snapshot() operation. As we have seen, p_i can benefit from the help provided by p_i by returning the array that p_j stored in $REG[j]$ at the end of its invocation of the operation update().

Theorem 34 *The algorithms described in Fig. 8.6 define a bounded wait-free implementation of an atomic snapshot object.*

Proof Let us first show that the implementation is bounded wait-free. As any invocation of update() contains an invocation of snapshot(), we have to show that any

```
operation update($v$) is
(1)  $help\_array_i \leftarrow$ snapshot();
(2)  $sn_i \leftarrow sn_i + 1$;
(3)  $REG[i] \leftarrow \langle v, sn_i, help\_array_i \rangle$;
(4)  return()
end operation.

operation snapshot() is
(5)  $can\_help_i \leftarrow \emptyset$;
(6)  $aa_i \leftarrow$ collect();
(7)  repeat forever
(8)      $bb_i \leftarrow$ collect();
(9)      if $(\forall j \in \{1, \dots, n\} : \ aa_i[j].sn = bb_i[j].sn)$
(10)        then return($aa_i[1..n].val$)
(11)        else for each $j \in \{1, \dots, n\}$ do
(12)                if $(aa_i[j].sn \neq bb_i[j].sn)$ then
(13)                   if $(j \in can\_help_i)$
(14)                        then return($bb_i[j].help\_array$)
(15)                        else $can\_help_i \leftarrow can\_help_i \cup \{j\}$
(16)                   end if
(17)                end if
(18)             end for
(19)     end if;
(20) $aa_i \leftarrow bb_i$
(21) end repeat
end operation.

internal operation collect() is
(22) for each $j \in \{1, \dots, n\}$ do $reg[j] \leftarrow REG[j]$ end for;
(23) return($reg$)
end operation.
```

Fig. 8.6 Bounded wait-free implementation of a snapshot object (code for p_i)

invocation of the snapshot() operation issued by a correct process terminates after a bounded number of steps (accesses to atomic registers).

Let us consider that p_i has not returned after having executed he **repeat** loop (lines 7n–21) n times. This means that, each time it has executed that loop, p_i found an identity j such that $aa_i[j].sn \neq bb_i[j].sn$ (line 9), which means that the corresponding process p_j issued a new invocation of the update() operation between the last two collect() issued by of p_i. Each time this occurs, the corresponding process identity j is a new identity added to the set can_help_i at line 15. (If j was already present in can_help_i, p_i would have executed line 14 instead of line 15, and would have consequently terminated its invocation of the snapshot() operation).

As by assumption p_i executes the loop n times, it follows that we have $can\_help_i = \{1, 2, \dots, n\} \setminus \{i\}$; i.e., this set contains the identities of other processes. It follows that, when it executes the loop for nth time and the test at line 9 is false, whatever the processes p_j such that $aa_i[j].sn \neq bb_i[j].sn$ at line 12, we necessarily have $j \in can\_help_i$ at line 13, from which it follows that p_i returns at line 14. The implementation is consequently wait-free, as p_i terminates after a finite number of operations on base registers have been executed.

Let us now replace "finite" by "bounded", i.e., let us determine a bound on the number of accesses to base registers. A collect() costs $O(n)$ accesses to base registers. The cost of each iteration of the **for** loop (lines 11–18) is $O(1)$, and there are at most n iteration steps, which means that the cost of that loop is upper-bounded by $O(n)$. Finally, as the enclosing **repeat** loop is executed at most n times, it follows that a process issues at most $O(n^2)$ accesses to base registers when it executes a snapshot() or an update() operation.

Let us now show that the object that is built is atomic. To that end we have to show that, for each execution (and according to the notation introduced in Chap. 4), there is a total order on the invocations of $\widehat{S}$ and update() and snapshot() such that: (1) $\widehat{S}$ includes all the invocations issued by the processes, except possibly, for each process, its the last invocation if that process crashes, (2) $\widehat{S}$ respects the real-time occurrence order on these invocations (i.e., if the invocation op$_1$ terminates before the invocation op$_2$ starts, op$_1$ has to appear before op$_2$ in $\widehat{S}$), and (3) $\widehat{S}$ respects the semantics of each operation (i.e., a snapshot() invocation has to return, for each process p_j, the value v_j such that, in $\widehat{S}$, there is no upadte() invocation by p_j between update(v_j) and that snapshot() invocation).

The definition of the sequence $\widehat{S}$ relies on (a) the atomicity of the base registers and (b) the fact that the operation snapshot() invokes *sequentially* the underlying function collect(). Let us remember that item (a) means that the read and write operations on the base registers can be considered as being executed instantaneously, each one at a point of the time line, and no two of them at the same time.

The sequence $\widehat{S}$ is built as follows. The linearization point of an invocation of the operation update() is the time at which it atomically executes the write in the corresponding SWMR register (line 3).

The definition of the linearization point of an invocation of the operation snapshot() depends on the line at which it returns:

- The linearization point of an invocation of snapshot() that terminates at line 10 (successful double collect) is at any time time between the end of the first and the beginning of the second of these collect invocations (see Theorem 33 and Fig. 8.4).
- The linearization point of an invocation of snapshot() that terminates at line 14 (i.e., p_i terminates with the help of another process p_j) is defined inductively as follows. (See Fig. 8.7, where a rectangle below an update() invocation represents the internal invocation of snapshot(). The dotted *help_array* arrow shows the way an array is conveyed from a successful double collect by a process p_k until an invocation of snapshot() issued by a process p_i.)

  The array (say *help_array*) returned by p_i was provided by an invocation of update() executed by some process p_j. As already seen, this update() was entirely executed within the time interval of p_i's current invocation of snapshot(). This

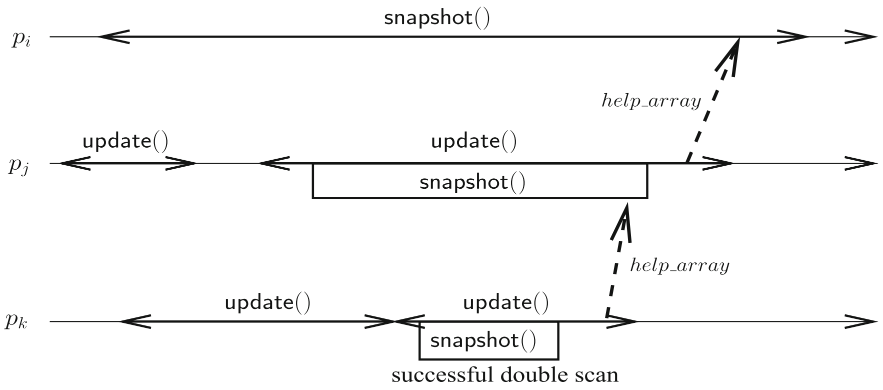


Fig. 8.7 Linearization point of an invocation of the snapshot() operation (case 2)

array was obtained by p_j from a successful double collect, or from another process p_k. If it was obtained from a process p_k, let us consider the way *help_array* was obtained by p_k. As there are at most n concurrent invocations of snapshot(), it follows by induction that there is a process p_x that has invoked the snapshot() operation and has obtained *help_array* from a successful double collect. Moreover, that invocation of snapshot() was inside an invocation of an update() operation that was entirely executed within the time interval of p_i's current invocation of snapshot().

The linearization point of the invocation of snapshot() issued by p_i is defined from the internal invocation of snapshot() whose successful double collect determined *help_array*. If several invocations of snapshot() are about to be linearized at the same time, they are ordered according to the total order in which they were invoked.

It follows directly from the previous definition of the linearization points associated with the invocations of the update() and snapshot() operation issued by the processes that $\widehat{S}$ satisfies items (1) and (2) stated at the beginning of the proof. The satisfaction of item (3) comes from the fact that the array returned by any invocation of snapshot() has always been obtained from a successful double collect. □

8.2.3 One-Shot Single-Writer Snapshot Object: Containment Property

A one-shot single-writer snapshot object is a single-writer snapshot object such that each process invokes each operation at most once and always invokes update() before snapshot().

As far as notations are concerned, let v_x be the value written by p_x and $snap_x$ be the array that it later obtained from its invocation of snapshot(). Let us remember that, if a process p_y has not yet invoked update() when $snap_x$ is returned, we have $snap_x[y] = \perp$. Finally let $(snap_x \preceq snap_y) \equiv \forall i : \big((snap_x[i] \neq \perp) \Rightarrow (snap_x[i] = snap_y[j] = v_j)\big)$. It follows from its atomicity property (linearization points) that a one-shot single-writer snapshot object satisfies the following properties:

- Self-inclusion. For any process p_x: $snap_x[x] = v_x$.
- Containment. For any two processes p_x and p_y: $(snap_x \preceq snap_y) \vee (snap_y \preceq snap_y)$.

Said differently, a one-shot single-writer snapshot object guarantees that all arrays which are returned are totally ordered by the $\preceq$ relation. This containment property is particularly useful and motivated the definition of the one-shot snapshot object.

8.3 Single-Writer Snapshot Object with Infinitely Many Processes

The following implementation is due to M.K. Aguilera (2004).

Computation model The computation model considered in this section is the *finite concurrency model* introduced in Sect. 7.1.2. There are infinitely many processes, but in each time interval only finitely many processes execute operations. Each process p_i has an identity i (an integer), and it is common knowledge that no two distinct process have the same identity. Moreover, the processes have not necessarily consecutive identities.

Adapting the internal representation: the array REG To adapt the previous snapshot construction to the finite concurrency model, we first consider that the array REG of SWMR atomic registers is a potentially infinite array, starting at $REG[1]$, so that an entry $REG[i]$ is associated with each "potential" process p_i. For each register $REG[i]$, the fields $REG[i].sn$ and $REG[i].val$ are initialized to 0 and $\perp$, respectively, where $\perp$ is a default value that no process can write.

Adapting the algorithm for the operation update() Now only the entries $REG[i]$ that do correspond to a process p_i that has invoked at least once the operation update() are meaningful. This means that, when a process p_i invokes update(), it has first to indicate in one way or another that, from now on, the entry $REG[i]$ is meaningful.

A simple way to do this consists in using a weak counter, denoted $WEAK\_CT$ as defined in Sect. 7.1.2. This idea is the following: $WEAK\_CT$ records the highest identity of a process that has invoked the operation update(). In that way, we know that the meaningful entries of the array REG are a subset of the entries $REG[j]$ such that $1 \leq j \leq WEAK\_CT$. More precisely, these entries correspond to the following set of (process identity, value) pairs:

$$\{(j, REG[j]) \mid (1 \leq j \leq WEAK\_CT) \;\wedge\; REG[j].val \neq \perp)\,\}.$$

The algorithm implementing the update() operation we obtain is described in Fig. 8.8. The lines with the same number in this figure and in the base algorithms

```
operation update($v$) is
(N0)    while ($WEAK\_CT$.get_count() $< i$) do $WEAK\_CT$.increment() end while;
(1)     $help\_array_i \leftarrow$ snapshot();
(2)     $sn_i \leftarrow sn_i + 1$;
(3)     $REG[i] \leftarrow \langle v, sn_i, help\_array_i \rangle$;
(4)     return()
end operation.

operation snapshot() is
(5)     $can\_help_i \leftarrow \emptyset$;
(M.6)   $n\_init \leftarrow WEAK\_CT$.get_count(); $aa_i \leftarrow$ collect($n\_init$);
(7)     repeat forever
(M.8)       $n \leftarrow WEAK\_CT$.get_count(); $bb_i \leftarrow$ collect($n$);
(9)         if ($\forall j \in \{1, \dots, n\}$ :  $aa_i[j].sn = bb_i[j].sn$)
(10)          then return($aa_i[1..n].val$)
(11)          else  for each $j \in \{1, \dots, n\}$ do
(12)                    if ($aa_i[j].sn \neq bb_i[j].sn$) then
(M.13)                    if $\big((j \in can\_help_i) \ \vee \ (j > n\_init)\big)$
(14)                          then return($bb_i[j].help\_array$)
(15)                          else  $can\_help_i \leftarrow can\_help_i \cup \{j\}$
(16)                      end if
(17)                    end if
(18)                end for
(19)        end if;
(20)    $aa_i \leftarrow bb_i$
(21)    end repeat
end operation.

internal operation collect($x$) is
(M.22)  for each $j \in \{1, \dots, x\}$ do $reg[j] \leftarrow REG[j]$ end for;
(23)    return($reg$)
end operation.
```

Fig. 8.8 Single-writer atomic snapshot for infinitely many processes (code for p_i)

of Fig. 8.6 are the same. As far as the operation update() is concerned, the only difference with respect to the base version is the addition of the first line marked N0.

Adapting the algorithm for the operation snapshot() A first problem that has to be solved consists in making the collect() function always terminate. A simple solution consists in adding an input parameter x to that function, indicating that the collect has to be only from $REG[1]$ until $REG[x]$. The value of this parameter is defined as the current value of the counter $WEAK\_CT$. The corresponding adaptation of the algorithm implementing the snapshot() operation appears in the lines prefixed by "M" (for modified); more precisely, the lines M.6, M.8, M.13, and M.22 in Fig. 8.8.

A second problem arises when new processes with higher identities invoke update(), causing the counter $WEAK\_CT$ to increase forever. It is consequently possible that, while it executes the **repeat** loop, an invocation of snapshot() never

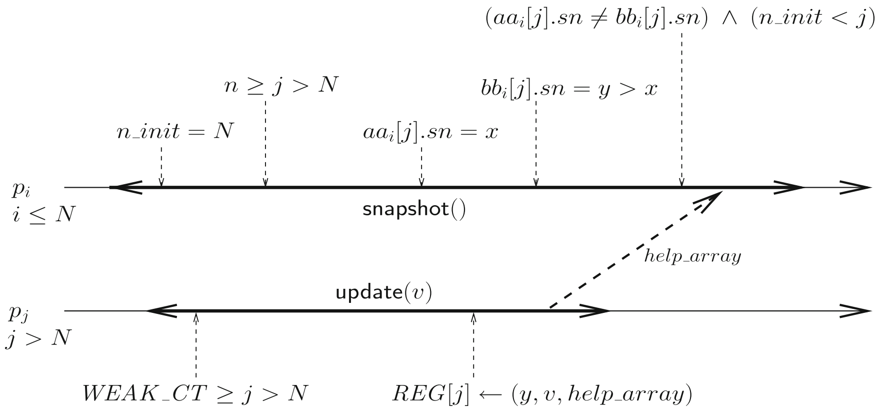


Fig. 8.9 An array transmitted from an update() to a snapshot() operation

finds a process p_j that has terminated two invocations of update() during its invocation of snapshot(): permanently, there are new invocations of update(), but those are issued by new processes with higher and increasing identities.

To solve this problem, let us observe that, if *WEAK_CT* increases due to a process p_j, then p_j has necessarily increased it (at line M0 when it executed the update() operation) after p_i started its snapshot operation. So, if n_init is the value of *WEAK_CT* when p_i starts invoking snapshot() (see line M.6), this means that we have $j > n\_init$. The solution to the problem (see Fig. 8.9) consists then in replacing the test $j \in could\_help_i$ by the test $j \in could\_help_i \vee j > n\_init$ (line M.3): even if p_j has not executed two update(), $REG[j].help\_array$ can be returned as it was determined after p_i started its invocation of the snapshot() operation.

Remarks As it is written, the returned value (at line 10 or 14) is an array that can contain lots of $\perp$. This depends on the identity of the processes that have previously invoked the update() operation. It is possible to return instead a set of (process identity, value) pairs. On another side, the array can be replaced by a list.

The proof that this is a wait-free implementation of an atomic snapshot object in the finite concurrency model is left as an exercise. The reader can easily remark that the construction is not bounded wait-free (this is because it is not possible a priori to state a bound on the number of iterations of the **while** loop).

8.4 Multi-Writer Snapshot Object

This section presents a multi-writer snapshot algorithm due to D. Imbs and M. Raynal (2011). This implementation is based on a helping mechanism similar to the one used in the previous section. The snapshot object has m components.

8.4.1 The Strong Freshness Property

When we look at the implementation of the operation update(v) described in Fig. 8.6, it appears that the values of the helping array saved by a process p_i in $REG[i].help\_array$ have been written before the write of v into $REG[i]$ (lines 1 and 3 of Fig. 8.6). It follows that, if this array of values is returned at line 13 of Fig. 8.6 by a process p_j, the value $help\_array[i]$ obtained by p_j is older than the value v.

We consider here the following additional property for a multi-writer snapshot object:

- Strong freshness. An invocation of snapshot() which is helped by an operation update(x, v) returns a value for the component x that is at least as recent as v.

The aim of this property is to provide every invocation of the operation snapshot with an array of values that are "as fresh as possible". As we will see, this property will be obtained by separating, inside the operation ypdate(x, v), the write v into $REG[i]$ from the write of the helping array. The corresponding strategy is called "write first, help later". (On the contrary, the implementation described in Fig. 8.6 is based on the strategy of computing a helping array first and later writing atomically both the value v and the helping array).

8.4.2 An Implementation of a Multi-Writer Snapshot Object

Internal representation of the multi-writer snapshot object The internal representation is made up of two arrays of atomic registers. Let us recall that m is the number of components of the snapshot object while n is the number of processes:

- The first array, denoted $REG[1..m]$, is made up of MWMR atomic registers. The register $REG[x]$ is associated with component x. It has three fields $\langle val, pid, sn\rangle$ whose meaning is the following: $REG[x].val$ contains the current value of the component x, while $REG[x].(pid, sn)$ is the "identity" of v. $REG[x].pid$ is the index of the process that issued the corresponding update(x, v) operation, while $REG[x].sn$ is the sequence number associated with this update when considering all updates issued by p_{pid}.
- The second array, denoted $HELPSNAP[1..n]$, is made up of one SWMR atomic register per process. $HELPSNAP[i]$ is written only by p_i and contains a snapshot value of $REG[1..m]$ computed by p_i during its last update() invocation. This snapshot value is destined to help processes that issued snapshot() invocations concurrent with p_i's update. More precisely, if during its invocation of snapshot() a process p_j discovers that it can be helped by p_i, it returns the value currently kept in $HELPSNAP[i]$ as output of its own invocation of snapshot().

The algorithm implementing the operation update(x, v) The algorithm implementing this operation is described at lines 1–4 of Fig. 8.10. It is fairly simple. Let

```
operation update$(x, v)$ is
(1)  $sn_i \leftarrow sn_i + 1$;
(2)  $REG[x] \leftarrow \langle v, i, sn_i \rangle$;
(3)  $HELPSNAP[i] \leftarrow$ snapshot();
(4)  return()
end operation.

operation snapshot() is
(5)  $can\_help_i \leftarrow \emptyset$;
(6)  for each $x \in \{1, \cdots, m\}$ do $aa[x] \leftarrow REG[x]$ end for;
(7)  repeat forever
(8)     for each $x \in \{1, \cdots, m\}$ do $bb[x] \leftarrow REG[x]$ end for;
(9)     if $(\forall x \in \{1, \cdots, m\} : aa[x] = bb[x])$ then return$(bb[1..m].val)$ end if;
(10)    for each $x \in \{1, \cdots, m\}$ such that $bb[x] \neq aa[x]$ do
(11)       let $\langle -, w, - \rangle = bb[x]$;
(12)       if $(w \in can\_help_i)$ then return$(HELPSNAP[w])$
(13)                                  else $can\_help_i \leftarrow can\_help_i \cup \{w\}$
(14)       end if
(15)    end for;
(16)    $aa \leftarrow bb$
(17) end repeat
end operation.
```

Fig. 8.10 Wait-free implementation of a multi-writer snapshot object (code for p_i)

p_i be the invoking process. First, p_i increases the local sequence number generator sn_i (initialized to 0) and atomically writes the triple $\langle v, i, sn_i \rangle$ into $REG[x]$. It then computes a snapshot value and writes it into $HELPSNAP[i]$ (line 3).

This constitutes the "write first, help later" strategy. The write of the value v into the component x is executed before the computation and the write of a helping array. The way $HELPSNAP[i]$ can be used by other processes was described previously. Finally, p_i returns from its invocation of update().

It is important to notice that, differently from what is done in Fig. 8.6, the write of v into $REG[x]$ and the write of a snapshot value into $HELPSNAP[i]$ are distinct atomic writes (which access different atomic registers).

The algorithm implementing the operation snapshot(): try first to terminate without help from a successful double collect This algorithm is described at lines 5–17 of Fig. 8.10.

The pair of lines 6 and 8 and the pair of lines 16 and 8 constitute "double collects". Similarly to what is done in Fig. 8.6, a process p_i first issues a double collect to try to compute a snapshot value by itself. The values obtained from the first collect are saved in the local array aa, while the values obtained from the second collect are saved in the local array bb. If $aa[x] = bb[x]$ for each component x, p_i has executed a successful double collect: $REG[1..m]$ contained the same values at any time during the period starting at the end of the first collect and finishing at the beginning of the second collect. Consequently, p_i returns the array of values $bb[1..m].val$ as the result of its snapshot invocation (line 9).

The algorithm implementing the operation snapshot()**: otherwise, try to benefit from the help of other processes** If the predicate $\forall x : aa[x] = bb[x]$ is false, p_i looks for all entries x that have been modified during its previous double collect. Those are the entries x such that $aa[x] \neq bb[x]$. Let x be such an entry. As witnessed by $bb[x] = \langle -, w, - \rangle$, the component x has been modified by process p_w (line 11).

The predicate $w \in can\_help_i$ (line 12) is the helping predicate. It means that process p_w issued two updates that are concurrent with p_i's current snapshot invocation. As we have seen in the algorithm implementing the operation update(x, v) (line 3; see also Fig. 8.11), this means that p_w has issued an invocation of snapshot() as part of an invocation of update() concurrent with p_i's snapshot invocation. If this predicate is true, the corresponding snapshot value (which has been saved in $HELPSNAP[w]$) can be returned by p_i as output of its snapshot invocation (line 12).

If the predicate is false, process p_i adds the identity w to the set can_help_i (line 13). Hence, can_help_i (which is initialized to $\emptyset$, line 1) contains identities y indicating that process p_y has issued its last update while p_i is executing its snapshot operation. Process p_i then moves the array bb into the array aa (line 16) and re-enters the **repeat**. (As already indicated, the lines 16 and 08 constitute a new double scan.)

On the "write first, help later" strategy As we can see, this strategy is very simple. It has several noteworthy advantages:

- This strategy first allows atomic write operations (at line 2 and line 3) to write values into base atomic registers $REG[r]$ and $HELPSNAP[i]$ that have a smaller size than the values written in the single-writer snapshot object implementation of Fig. 8.6 (where an atomic write into $REG[x]$ is on a triple of values). Atomic writes of smaller values allow for more efficient solutions.
- Second, this simple strategy allows the atomic writes into the base atomic registers $REG[x]$ and $HELPSNAP[i]$ to be not synchronized (while they are strongly synchronized in the single-writer snapshot implementation of Fig. 8.6, where they are pieced into a single atomic write).

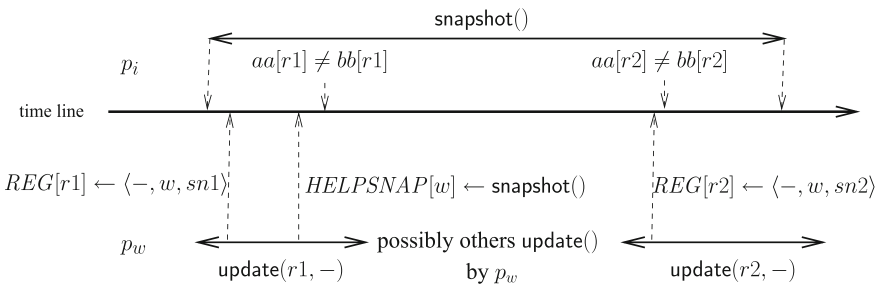

snapshot()
p_i
$aa[r1] \neq bb[r1]$
$aa[r2] \neq bb[r2]$
time line
$REG[r1] \leftarrow \langle -, w, sn1 \rangle$
$HELPSNAP[w] \leftarrow$ snapshot()
$REG[r2] \leftarrow \langle -, w, sn2 \rangle$
p_w
possibly others update() by p_w
update$(r1, -)$
update$(r2, -)$

Fig. 8.11 A snapshot() with two concurrent update() by the same process

- Finally, as shown in the proof, the "write first, help later" strategy allows the invocations of snapshot() to satisfy the strong freshness property (i.e., to return component values that are "as fresh as possible").

Cost of the implementation This section analyses the cost of the operations update() and snapshot() in terms of the number of base atomic registers that are accessed by a read or write operation.

- Operation snapshot().
  - Best case. In the best case an invocation of the operation snapshot() returns after having read only twice the array $REG[1..m]$. The cost is then $2m$.
  - Worst case. Let p_i be the process that invoked operation snapshot(). The worst case is when a process returns at line 12 and the local array can_help_i contains $n-1$ identities: an identity from every process but p_i. In that case, p_i has read $n+1$ times the array $REG[1..m]$ and, consequently, has accessed $(n+1)m$ times the shared memory.
- The cost of an update operation is the cost of a snapshot operation plus 1.

It follows that the cost of an operation is $O(n \times m)$.

8.4.3 Proof of the Implementation

Definition 1 The array of values $[v_1, \ldots, v_m]$ returned by an invocation of snapshot() is *well defined* if, for each x, $1 \le x \le m$, the value v_x has been read from $REG[x]$.

Definition 2 The values returned by an invocation of snapshot() are *mutually consistent* if there is a time at which they were simultaneously present in the snapshot object.

Definition 3 The values returned by an invocation of snapshot() are strongly fresh if, for each x, $1 \le x \le m$, the value v_x returned for component x is not older than the last value written into $REG[x]$ before the snapshot invocation. (Let us recall that, as each $REG[x]$ is an atomic register, its read and write operations can be totally ordered in a consistent way. The term "last" is used with respect to this total order).

Definition 4 Let snap be an invocation of snapshot() issued by a process p_i.

- The invocation snap is 0-helped if it terminates with a successful double collect (line 9 of Fig. 8.10).
- The invocation snap is 1-helped if it terminates by returning $HELPSNAP[w1]$ (line 12 of Fig. 8.10) and the values in $HELPSNAP[w1]$ come from a successful double collect by p_{w1} (i.e., the values in $HELPSNAP[w1]$ have been computed at line 3 by the invocation of snapshot() inside the invocation of update() issued by p_{w1}).

- The invocation snap is 2-helped if it terminates by returning $HELPSNAP[w1]$ (line 12) and the values in $HELPSNAP[w1]$ come from a 1-helped snapshot() operation invoked by a process p_{w2} at line 03.
- For the next values of h, the h-helped notion is similarly defined by induction.

Lemma 14 *The values returned by a* 0*-helped invocation of* snapshot() *are well defined, mutually consistent, and strongly fresh.*

Proof Let us consider a 0-helped invocation of the operation snapshot(). As it terminates at line 9, the array of values returned are the values in the array $bb[1..m]$. It follows from line 8 that this array is well defined (its values are from the array $REG[1..m]$).

Mutual consistency and strong freshness follow from the fact that the termination of the invocation of snapshot() is due to a successful double collect. More precisely, the values kept in $bb[1..m]$ were present simultaneously in $REG[1..m]$ during the period starting at the end of the first collect and ending at the beginning of the second collect (mutual consistency) and, for any entry x, the value returned from $REG[x]$ is not older than the value kept in $REG[x]$ when the invocation of snapshot() started (strong freshness). □

Lemma 15 *The values returned by a* 1*-helped* snapshot() *operation are well defined, mutually consistent, and strongly fresh.*

Proof Let snap be a 1-helped invocation of snapshot(). It follows from the definition of 1-helped invocation that (a) the array returned by snap (namely the value read by p_i from $HELPSNAP[w1]$) was computed by a snapshot operation snap′ invoked inside an invocation of update() issued by p_{w1}, and (b) snap′ is 0-helped (i.e., $HELPSNAP[w1]$ comes from a successful double collect). It then follows from Lemma 14 that the values in $HELPSNAP[w1]$ are well defined and mutually consistent.

Let us now show that the values contained in $HELPSNAP[w1]$ are strongly fresh; i.e., for each x, the value in $HELPSNAP[w1][x]$ is not older than the last value written into $REG[x]$ before snap started. Let up1 and up2 be the two updates issued by p_{w1} whose sequence numbers are $sn1$ and $sn2$ (let us observe that, due to the test of line 12, these updates do exist). Moreover, snap′ is an internal invocation issued by an invocation up′ of update() from p_{w1} whose sequence number sn is such that $sn1 \leq sn \leq sn2$.

As the value of $HELPSNAP[w1]$ that is returned is computed during up′, and up′ was invoked by p_{w1} after up1, the values obtained from REG and assigned to $HELPSNAP[w1]$ were read after up1 started. We claim that up1 started after snp. It follows from this claim that the values in $HELPSNAP[w1]$ that are returned were read from REG after snap started, which proves the lemma.

Proof of the claim. Let $r1$ be the component that entailed the addition of the identity $w1$ to can_help_i at line 13 during the execution of snap. Moreover, let $aa1[r1]$ and $bb1[r1]$ be the values in the local arrays aa and bb that entailed this addition. Hence, we have $aa1[r1] \neq bb1[r1]$ (line 10) and $bb1[r1] = \langle -, w1, - \rangle$.

As snap has read $REG[r1]$ twice (first to obtain $aa1[r1]$, and then to obtain $bb1[r1]$) and $aa1[r1] \neq bb1[r1]$, it follows that up1 started after the start of snp, which concludes the proof of the claim. □

Lemma 16 *For any h, the values returned by an h-helped invocation of* snapshot() *are well-defined, mutually consistent, and strongly fresh.*

Proof The proof is by induction. The base case is $h = 0$ (Lemma 14). Assuming that the lemma is satisfied up to $h - 1$, the proof for h is similar to the proof for $h = 1$ that relies on the fact that we have a proof for the case $h - 1 = 0$ (Lemma 15). □

Lemma 17 *Wait-freedom. Any invocation of* update() *or* snapshot() *issued by a correct process terminates.*

Proof Let us first observe that, if every snapshot() operation issued by a correct process terminates, then all its update() operations terminate. Hence, the proof only has to show that all snapshot() operations issued by a correct process terminate.

Let us consider an invocation of snapshot() issued by a correct process p_i. If, when p_i executes line 9, the predicate is true, the invocation terminates. So, we have to show that, if the predicate of line 9 is never satisfied, then the predicate of line 12 eventually becomes true. As the predicate of line 9 is never satisfied, each time p_i executes the loop body, there is a component x such that $aa[x] \neq bb[x]$. The process p_k that modified $REG[k]$ between the two readings by p_i entails the addition of its identity k to can_help_i (where k is extracted from $bb[x]$). In the worst case, $n - 1$ identities (one per process except p_i, because it cannot execute an update operation while it executes a snapshot operation) can be added to can_help_i while the predicate of line 12 remains false. But, once can_help_i contains one identity per process (but p_i), the test of line 12 necessarily becomes satisfied, which proves the lemma. □

The next lemma shows that the implementation is atomic, i.e., that the invocations of update() and snapshot() issued by the processes during a run (except possibly the last operation issued by faulty processes) appear as if they have been executed one after the other, each one being executed at some point of the time line between its start event and its end event.

Lemma 18 *The algorithms described in Fig. 8.10 implement an atomic multi-writer snapshot object.*

Proof The proof consists in associating with each invocation inv of update() and snapshot() a single point of the time line denoted ℓp(inv) (linearization point of inv) such that:

- ℓp(inv) lies between the beginning (start event) of inv and its end (end event),
- no two operations have the same linearization point,
- the sequence of the operation invocations defined by their linearization points is a sequential execution of the snapshot object.

So, the proof consists in (a) an appropriate definition of the linearization points and (b) showing that the associated sequence satisfies the specification of the snapshot object.

- Point (a): definition of the linearization points.

  The linearization point of each operation invocation (except possibly the last operation of faulty processes) is defined as follows:

  – The linearization point of an invocation of update(r, $-$) is the linearization point of its write of $REG[r]$ (line 2). (Let us remember that, as the underlying atomic registers are atomic, they have well-defined linearization points).

  – The linearization point of an invocation psp of snapshot() depends on the line at which the return() statement is executed:

    * Case 1: psp returns at line 9 due a successful double collect (i.e., psp is 0-helped). Its linearization point is any point of the time line between the first and the second collect of that successful double collect.

    * Case 2: psp returns at line 12 (i.e., psp is h-helped with $h \geq 1$). In this case, the array of values returned by psp was computed by some update operation at line 3. Moreover, whatever the value of h, this array was computed by a successful double collect executed by some process p_z. When considering this successful double collect, ℓp(psp) is placed between the end of its first collect and the beginning of its second collect.

  If two operations are about to be linearized at the same point, they are arbitrarily ordered (e.g., according to the identities of the processes that issued them).
  It follows from the previous linearization point definitions that each invocation of an operation is linearized between its beginning and its end, and no two operations are linearized at the same point.

- Point (b): the sequence of invocations of update() and snapshot() defined by their linearization points satisfies the specification of the snapshot object.
  This follows directly from Lemma 15 which showed that the values returned by every snapshot operation are well defined, mutually consistent, and strongly fresh. □

Theorem 35 *The implementation described in Fig. 8.10 is a bounded wait-free implementation of an atomic multi-writer snapshot object which also satisfies the strong freshness property.*

Proof The proof that the implementation satisfies the consistency property of a snapshot object follows from Lemma 18. Wait-freedom follows from Lemma 17. The freshness property follows from the definition of the linearization points given in Lemma 18. Finally, the fact that the implementation is bounded wait-free follows from the fact that an operation costs at most $O(m \times n)$ accesses to base atomic registers. □

8.5 Immediate Snapshot Objects

The notion of an immediate snapshot object is due to E. Borowsky and E. Gafni (1993).

8.5.1 One-Shot Immediate Snapshot Object: Definition

A one-shot *immediate snapshot* object is a one-shot snapshot object where the update() and snapshot() are fused in a single operation denoted update_snapshot() and such that each process invokes at most once that operation. (A similar fusion of write and read operations has already been done in Sect. 7.3, where the operations store() and collect() were pieced together to give rise to the operation store_collect()).

When a process p_i invokes update_snapshot(v), it deposits v and obtains a view $view_i$ made up of (process identity, value) pairs. From an external observer point of view, everything has to appear as if the operation was executed instantaneously.

More formally, a one-shot immediate snapshot object is defined by the following properties. Let $view_i$ denote the set of (process identity, value) pairs returned by p_i when it invokes update_snapshot(v_i).

- Liveness. An invocation of update_snapshot() by a correct process terminates.
- Self-inclusion. $(i, v_i) \in view_i$.
- Containment. $\forall i, j : view_i \subseteq view_j$ or $view_j \subseteq view_i$.
- Immediacy. $\forall i, j$: if $(j, v_j) \in view_i$ then $view_j \subseteq view_i$.

The first three properties (liveness, self-inclusion, and containment) are the properties which define a one-shot snapshot object. When considering only these properties, update_snapshot(v_i) could be implemented by the invocation update(v_i) immediately followed by the invocation snapshot().

The immediacy property is not satisfied by this implementation, which shows the fundamental difference between snapshot and immediate snapshot.

8.5.2 One-Shot Immediate Snapshot Versus One-Shot Snapshot

Figure 8.12 depicts three processes p_1, p_2, and p_3. Each process p_i first invokes update(v_i) and later invokes snapshot(). (In the figure, process identities appear as subscripts in the operation invoked).

According to the specification of the one-shot snapshot object, the invocation snapshot$_1$() returns the view $\{(1, v_1), (2, v_2)\}$, while both the invocations snapshot$_2$() issued by p_2 and snapshot$_3$() issued by p_3 return the view $\{(1, v_1), (2, v_2), (3, V_3)\}$. This means that it is possible to associate with this execution the following sequence of operation invocations $\widehat{S}$:

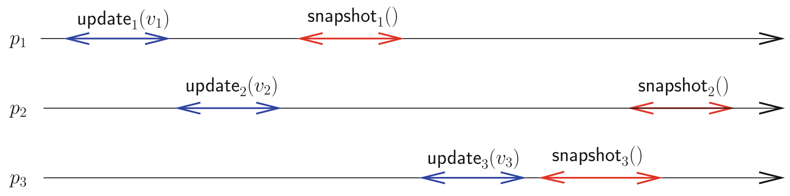


Fig. 8.12 An execution of a one-shot snapshot object

$$\mathsf{update}_1(v_1)\ \mathsf{update}_2(v_2)\ \mathsf{snapshot}_1()\ \mathsf{update}_3(v_3)\ \mathsf{snapshot}_2()\ \mathsf{snapshot}_3(),$$

which belongs to the sequential specification of a one-shot snapshot object and shows, consequently, that this execution is atomic.

Figure 8.13 shows the same processes where the invocations $\mathsf{update}_i()$ and $\mathsf{snapshot}_i()$ issued by p_i are replaced by a single invocation $\mathsf{update\_snapshot}_i()$ (that starts at the same time as $\mathsf{update}_i()$ starts and terminates at the same time as $\mathsf{snapshot}_i()$).

As $\mathsf{update\_snapshot}_1(1)$ terminates before $\mathsf{update\_snapshot}_3(3)$ starts, we necessarily have $(1, v_1) \in view_3$ and $(3, v_3) \notin view_1$. More generally, each of the five items that follow defines a set of correct outputs for the execution of Fig. 8.13 (said differently, this execution is non-deterministic in the sense that its outputs are defined by any one of these five items):

1. $view_1 = \{(1, v_1)\}$, $view_2 = \{(1, v_1), (2, v_2)\}$, and $view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$,
2. $view_1 = view_2 = \{(1, v_1), (2, v_2)\}$, and $view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$,
3. $view_2 = \{(2, v_2)\}$, $view_1 = \{(1, v_1), (2, v_2)\}$, and $view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$,
4. $view_1 = \{(1, v_1)\}$, and $view_2 = view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$, and
5. $view_1 = \{(1, v_1)\}$, $view_3 = \{(1, v_1), (3, v_3)\}$, and $view_2 = \{(1, v_1), (2, v_2), (3, v_3)\}$.

When $view_1$, $view_2$ and $view_3$ are all different (item 1), everything appears as if the three invocations of $\mathsf{update\_snapshot}()$ have been executed sequentially (and

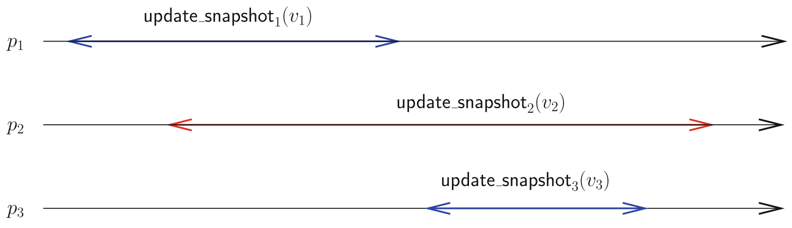


Fig. 8.13 An execution of an immediate one-shot snapshot object

consistently with their real-time occurrence order). When two of them are equal, e.g., $view_1 = view_2 = \{(1, v_1), (2, v_2)\}$ (item 2), everything appears as if the invocations update_snapshot$_1$(1) and update_snapshot$_2$(2) have been issued at the very same time, both before update_snapshot$_3$(). This possibility of simultaneity is the very essence of the "immediate" snapshot abstraction. It also shows that an immediate snapshot object is not an atomic object.

Theorem 36 *A one-shot immediate snapshot object satisfies the following property: if $(i, -) \in view_j$ and $(j, -) \in view_i$, then $view_i = view_j$.*

Proof If $(j, -) \in view_i$ (theorem assumption), we have $view_j \subseteq view_i$ from the immediacy property. Similarly, $(i, -) \in view_j$ implies that $view_i \subseteq view_j$. It trivially follows that $view_i = view_j$ when $(j, -) \in view_i$ and $(i, -) \in view_j$. $\square$

Set-linearizability The previous theorem states that, while its operations appear as if they were executed instantaneously, an immediate snapshot object is not an atomic object. This is because it is not always possible to totally order all its operations. The immediacy property states that, from a logical time point of view, it is possible that several invocations appear as being executed simultaneously (they then return the same view), making it impossible to consider that one occurred before the other. This means that an immediate snapshot object has no sequential specification. We then say that these invocations are *set-linearized* at the same point of the time line, and the notion of a linearization point is replaced by *set-linearization*.

Hence, differently from a snapshot object, the specification of an immediate snapshot object allows for concurrent operations from an external observer point of view. It then requires that the invocations which are set-linearized at the same point do return the very same view.

8.5.3 An Implementation of One-Shot Immediate Snapshot Objects

This section describes a one-shot immediate snapshot implementation due to E. Borowsky and E. Gafni (1993).

Underlying data structure This implementation is based on two arrays of SWMR atomic registers:

- The array $REG[1..n]$, which is initialized to $[\perp, \ldots, \perp]$, is such that $REG[i]$ is used by p_i to store its value v_i.
- $LEVEL[1..n]$ is initialized to $[n + 1, \ldots, n + 1]$. $LEVEL[i]$ can be written only by p_i. A process p_i uses also a local array $level_i[1..n]$ to keep the last values it (asynchronously) reads from $LEVEL[1..n]$. A register $LEVEL[i]$ contains at most n distinct values (from $n+1$ until 1), which means that it requires $b = \lceil \log_2(n+1) \rceil$ bits.

```
operation update_snapshot($v_i$) is
(1)  $REG[j] \leftarrow v_i$;
(2)  repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
(3)          for each $j \in \{1,\ldots,n\}$ do $level_i[j] \leftarrow LEVEL[j]$ end for;
(4)          $set_i \leftarrow \{x \mid level_i[x] \leq level_i[i]\}$
(5)  until $(|set_i| \geq level_i[i])$ end repeat;
(6)  let $view_i = \{ (x, REG[x]) \mid x \in set_i \}$;
(7)  return($view_i$)
end operation.
```

Fig. 8.14 An algorithm for the operation update_snapshot() (code for process p_i)

Underlying principles and the algorithm implementing the operation update_ snapshot() Let us consider the image of a stairway made up of $n+1$ stairs. Initially all the processes stand at the highest stair (i.e., the stair whose number is $n+1$). (This is represented in the algorithm by the initial values of the *LEVEL* array; namely, for any process p_j, we have $LEVEL[j] = n + 1$).

The algorithm, which is described in Fig. 8.14, is based on the following idea. When a process p_i invokes update_snapshot(), it first deposits its value v_i in $REG[i]$ (line 1). Then it descends along the stairway (lines 2–5), going from step $LEVEL[i]$ to step $LEVEL[i] - 1$ until it attains a step k such that there are exactly k processes (including itself) stopped on steps 1 to k. The identities of these k processes are saved in the local set set_i. It then returns the view made up of the k pairs $(x, REG[x])$ such that $x \in set_i$(the k processes which stopped on one of the steps 1 to k).

To catch the underlying intuition and understand how this idea works, let us consider two extremal cases in which k processes invoke the update_snapshot() operation:

- Sequential case.

  In this case, the k processes invoke the operation sequentially; i.e., the next invocation starts only after the previous one has returned. It is easy to see that the first process p_{i_1} that invokes the update_snapshot() operation proceeds from step $n + 1$ until step number 1, and stops at this step. Then, the process p_{i_2} starts and descends from step $n + 1$ until step number 2, etc., and the last process p_{i_k} stops at step k.

  Moreover, the set returned by p_{i_1} is $\{i_1\}$, the set returned by p_{i_2} is $\{i_1, i_2\}$, etc., the set returned by p_{i_k} being $\{i_1, i_2, \ldots, i_k\}$. These sets trivially satisfy the inclusion property.

- Synchronous case.

  In this case, the k processes proceed synchronously. They all, simultaneously, descend from step $n + 1$ to step n, then from step n to step $n - 1$, etc., and they all stop at step number k because there are then k processes at steps from 1 to k (they all are on the same kth step).

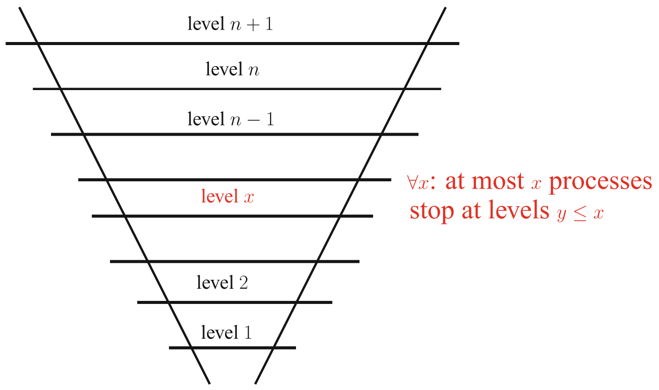


Fig. 8.15 The levels of an immediate snapshot objects

It follows that all the processes return the very same set of participating processes, namely, the set including all of them $\{i_1, i_2, \dots, i_k\}$, and the inclusion property is trivially satisfied.

Other cases, where the processes proceed asynchronously and some of them crash, can easily be designed. The general case is described in Fig. 8.15. If p_i stops at level x, its view includes all the pairs $\langle j, v_j \rangle$ such that p_j returns or crash at level x.

How levels are implemented The main question is now how to make this idea operational? This is done by three statements (Fig. 8.14). Let us consider a process p_i:

- First, when it is standing at a given step $LEVEL[i]$, p_i reads the steps at which the other processes are (line 3). The aim of this asynchronous reading is to allow p_i to compute an approximate global state of the stairway. Let us notice that, as a process p_j can only go downstairs, $level_i[j]$ is equal or smaller to the step $k = LEVEL[i]$ on which p_j currently is. It follows that, despite the fact the global state obtained by p_i is approximate, set_i can be safely used by p_i.
- Then (line 4), p_i uses the global state of the stairway it has obtained to compute the set (denoted set_i) of processes that, from its point of view, are standing at a step between $LEVEL][1]$ and $LEVEL][i]$ (the step where p_i currently is).
- Finally (line 5), if set_i contains $k = LEVEL][i]$ or more processes, p_i returns the corresponding view (line 6–7). Otherwise, it proceeds to the next stair $LEVEL]$ $[i] - 1$ (line 2).

Two preliminary lemmas are proved before the main theorem.

Lemma 19 *Let $set_i = \{x \mid level_i[x] \le LEVEL[i]\}$ (as computed at line 4). For any process p_i, the predicate $|set_i| \le LEVEL[i]$ is always satisfied at line 5.*

Proof Let us first observe that $level_i[i]$ and $LEVEL[i]$ are always equal at lines 4 and 5. Moreover, any $LEVEL[j]$ register can only decrease, and for any pair (i, j) we have $LEVEL[j] \leq level_i[j]$.

The proof is by contradiction. Let us assume that there is at least one process p_i such that $|set_i| = |\{x \mid level_i[x] \leq LEVEL[i]\}| > LEVEL[i]$. Let k be the current value of $LEVEL[i]$ when this occurs.$|set_i| > k$ and $LEVEL[i] = k$ mean that at least $k+1$ processes have progressed at least to stair k. Moreover, as any process p_j descends one stair at a time (it proceeds from stair $LEVEL[j]$ to stair $LEVEL[j]-1$ without skipping stairs),at least $k+1$ processes have proceeded from stair $k+1$ to stair k.

Among the at least $k+1$ processes that are on a stair $\leq k$, let p_ℓ be the last process that updated its $LEVEL[\ell]$ register to $k+1$ (due to the atomicity of the base registers, there is such a last process). When p_ℓ was on stair $k+1$ (we then had $LEVEL[\ell] = k+1$), it obtained at line 4 a set set_ℓ such that $|set_\ell| = |\{x \mid level_\ell[x]| \leq LEVEL[\ell]\} \geq k+1$ (this is because at least $k+1$ processes have proceeded to the stair $k+1$, and as p_ℓ is the last of them, it read a value smaller than or equal to $k+1$ from its own $LEVEL[\ell]$ register and the ones of those processes). As $|set_\ell| \geq k+1$, p_ℓ stopped descending the stairway at line 5, at stair $k+1$. It then returned, contradicting the initial assumption stating that it progresses until stair k. □

Lemma 20 *If p_i halts at stair k, we then have $|set_i| = k$. Moreover, set_i is composed of the processes that are at a stair $k' \leq k$.*

Proof Due to Lemma 19, we always have $|set_i| \leq LEVEL[i]$, when p_i executes line 5. If it stops, we also have $|set_i| \geq LEVEL[i]$ (test of line 5). It follows that $|set_i| = LEVEL[i]$. Finally, if k is p_i's current stair, we have $LEVEL[i] = k$ (definition of $LEVEL[i]$ and line 2). Hence, $|set_i| = k$.

The fact that set_i is composed of the identities of the processes that are at a stair smaller than or equal to k follows from the very definition of set_i (namely,$set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$), the fact that,for any x,$level_i[x] \leq LEVEL[x]$, and the fact that a process never climbs the stairway (it either halts on a stair,line 5, or descends to the next one, line 2). □

Theorem 37 *The algorithm described in Fig. 8.14 is a bounded wait-free implementation of a one-shot immediate snapshot object.*

Proof Let us observe that (1) $LEVEL[i]$ is monotonically decreasing, and (2) at any time, set_i is such that $|set_i| \geq 1$ (because it contains at least the identity i). It follows that the **repeat** loop always terminates (in the worst case when $LEVEL[i] = 1$). Hence, the algorithm is wait-free. Moreover, p_i executes the **repeat** loop at most n times, and each computation inside the loop includes n reads of atomic base registers. It follows that $O(n^2)$ is an upper bound on the number of read/write operations on base registers issued in an invocation of `update_snapshot()`. The algorithm is consequently bounded wait-free.

The self-inclusion property is a direct consequence of the way set_i is computed (line 4): trivially,the set $\{x | level_i[x] \leq level_i[i]\}$ always contains i.

For the containment property, let us consider two processes p_i and p_j that stop at stairs k_i, and k_j, respectively. Without loss of generality, let $k_i \leq k_j$. Due to Lemma 20, there are exactly k_i processes on the stairs 1 to k_i, and k_j processes on stairs 1 to $k_j \leq k_i$. As no process backtracks on the stairway (a process proceeds downwards or stops), the set of k_j processes returned by p_j includes the set of k_1 processes returned by p_i.

Let us finally consider the immediacy property. Let us first observe that a process deposits its value before starting its descent of the stairway (line 1), from which it follows that, if $j \in set_i$, $REG[j]$ contains the value v_j deposited by p_j. Moreover, it follows from lines 4 and 5 that, if a a process p_j stops at a stair k_j and then $i \in set_j$, then p_i stopped at a stair $k_i \leq k_j$. It then follows from Lemma 20 that the set set_j returned by p_j includes the set set_i returned by p_i, from which follows the immediacy property. □

8.5.4 A Recursive Implementation of a One-Shot Immediate Snapshot Object

This section describes a recursive implementation of a one-shot immediate snapshot object due to E. Gafni and S. Rajsbaum (2010). This construction can be seen as a recursive formulation of the previous iterative algorithm.

Underlying data structure As we are about to see, when considering distributed computing, an important point that distinguishes distributed recursion from sequential recursion on data structures lies in the fact that the recursion parameter is usually the number n of processes involved in the computation. The recursion parameter is used by a process to compute a view of the concurrency degree among the participating processes.

The underlying data structure representing the immediate snapshot object consists of a shared array $REG[1..n]$ such that each $REG[x]$ is an array of n SWMR atomic registers. The aim of $REG[x]$, which is initialized to $[\perp, \ldots, \perp]$, is to contain the view obtained by the processes that see exactly x other processes in the system. For any x, $REG[x]$ is accessed only by the processes that attain recursion level x and the atomic register $REG[x][i]$ can be read by all these processes but can be written only by p_i.

The recursive algorithm implementing the operation update_snapshot() The algorithm is described in Fig. 8.16. Its main call is an invocation of rec_update _snapshot(n, v_i), where n is the initial value of the recursion parameter and v_i the value that p_i wants to deposit into the immediate snapshot object (line 1). This call is said to occur at recursion level n. More generally, an invocation rec_update_snapshot(x, $-$) is said to occur at recursion level x. Hence, the recursion levels are decreasing from level n to level $n - 1$, then to level $n - 2$, etc. (Actually, a recursion level corresponds to what was called a "level" in Sect. 8.5.3.)

```
operation update_snapshot($v_i$) is
 (1)  $my\_view_i \leftarrow$ rec_update_snapshot($n, v_i$)
 (2)  return($my\_view_i$)
end operation.

operation rec_update_snapshot($x, v$) is
      % $x$ is the recursion parameter ($n \geq x \geq 1$) %
 (3)  $REG[x][i] \leftarrow v$;
 (4)  for each $j \in \{1, \ldots, n\}$ do $reg_i[j] \leftarrow REG[x][j]$ end for;
 (5)  $view_i \leftarrow \{\ (j, reg_i[j]) \mid reg_i[j] \neq \bot\ \}$;
 (6)  if ($|view_i| = x$) then $res_i \leftarrow view_i$
 (7)                      else $res_i \leftarrow$ rec_update_snapshot($x - 1, v$)
 (8)  end if;
 (9)  return($res_i$)
end operation.
```

Fig. 8.16 Recursive construction of a one-shot immediate snapshot object (code for process p_i)

When it invokes rec_update_snapshot(x, v), p_i first writes v into $REG[x][i]$ and reads asynchronously the content of $REG[x][1..n]$ (lines 3–4, let us notice that these lines implement a store-collect). Hence, the array $REG[x][1..n]$ is devoted to the xth recursion level.

Then, p_i computes the view $view_i$ obtained from $REG[x][1..n]$ (line 5). Let us remark that, as the recursion levels are decreasing and there are at most n participating processes, the set $view_i$ contains the values deposited by $n' = |view_i|$ processes, where n' is the number of processes that, from p_i's point of view, have attained recursion level x.

If p_i sees that exactly x processes have attained the recursion level x (i.e., $n' = x$), it returns $view_i$ as the result of its invocation of the immediate snapshot object (lines 6 and 9). Otherwise, fewer than x processes have attained recursion level x and consequently p_i invokes recursively rec_update_snapshot$(x - 1, v)$ (line 7) in order to attain a recursion level $x' < x$ accessed by exactly x' processes. It will stop its recursive invocations when it attains such a recursion level (in the worst case, $x' = 1$).

Theorem 38 *The algorithm described in Fig. 8.16 is a wait-free construction of an immediate snapshot object. Its step complexity (number of shared memory accesses) is $O(n(n - |res| + 1))$, where res is the set returned by* update_snapshot(v).

Proof Claim C. If at most x processes invoke rec_update_snapshot$(x, -)$ then (a) at most $(x - 1)$ processes invoke rec_update_snapshot$(x - 1, -)$ and (b) at least one process stops at line 6 of its invocation rec_update_snapshot$(x, -)$.

Proof of claim C. Assuming that at most x processes invoke update_snapshot $(x, -)$, let p_k be the last process that writes into $REG[x][1..n]$. We necessarily have $|view_k| \leq x$.If p_k finds $|view_k| = x$, it stops at line 6. Otherwise, we have $|view_k| < x$ and p_k invokes rec_update_snapshot$(x - 1, -)$ at line 7. But in that case, as p_k is the last process that wrote into the array $REG[x][1..n]$, it follows from $|view_k| < x$ that fewer than x processes have written into $REG[x][1..n]$, and consequently, at most $(x - 1)$ processes invoke rec_update_snapshot$(x - 1, -)$. End of the proof of claim C.

To prove the termination property, let us consider a correct process p_i that invokes update_snapshot(v_i). Hence, it invokes rec_update_snapshot$(n, -)$. It follows from Claim C and the fact that at most n processes invoke rec_update_snapshot $(n, -)$ that either p_i stops at that invocation or belongs to the set of at most $n - 1$ processes that invoke rec_update_snapshot$(n - 1, -)$. It then follows by induction from the claim that if p_i has not stopped during a previous invocation, it is the only process that invokes rec_update_snapshot(1). It then follows from the text of the algorithm that it stops at that invocation.

The proof of the self-inclusion property is trivial. Before stopping at recursion level x (line 6), a process p_i has written v_i into $REG[x][i]$ (line 3), and consequently we have then $(i, v_i) \in view_i$, which concludes the proof of the self-inclusion property.

To prove the self-containment and immediacy properties, let us first consider the case of two processes that return at the same recursion level x. If a process p_i returns at line 6 of recursion level x, let $view_i[x]$ denote the corresponding value of $view_i$. Among the processes that stop at recursion level x, let p_i be the last process which writes into $REG[x][1..n]$. As p_i stops, this means that $REG[x][1..n]$ has exactly x entries different from $\perp$ and (due to Claim C) no more of its entries will be set to a non-$\perp$ value. It follows that, as any other process p_j that stops at recursion level x reads x non-$\perp$ entries from $REG[x][1..n]$, we have $view_i[x] = view_j[x]$ which proves the properties.

Let us now consider the case of two processes p_i and p_j that return at line 6 of recursion level x and y, respectively, with $x > y$; i.e., p_i returns $view_i[x]$ while p_j returns $view_j[y]$. The self-containment follows then from $x > y$ and the fact that p_j has written into all the arrays $REG[z][1..n]$ with $n \geq z \geq y$, from which we conclude that $view_j[y] \subseteq view_i[x]$. Moreover, as $x > y$, p_i has not written into $REG[y][1..n]$ while p_j has written into $REG[x][1..n]$, and consequently $(j, v_j) \in view_i[x]$ while $(i, v_i) \notin view_j[y]$, from which the containment and immediacy properties follow.

As far as the number of shared memory accesses is concerned we have the following. Let res be the set returned by an invocation of rec_update_snapshot$(n, -)$. Each recursive invocation costs $n + 1$ shared memory accesses (lines 3–4). Moreover, the sequence of invocations, namely rec_update_snapshot$(n, -)$, rec _update_snapshot$(n - 1, -)$, etc., until rec_update_snapshot$(|res|, -)$ (where $x = |res|$ is the recursion level at which the recursion stops) contains $n - |res| + 1$ invocations. It follows that the cost is $O(n(n - |res| + 1))$ shared memory accesses. □

8.6 Summary

This chapter was on the notion of a snapshot object. It has shown how such an object can be wait-free implemented on top of base read/write registers despite asynchrony and any number of process crashes. It has also shown how an implementation for a fixed number of processes can be extended to cope with infinitely many processes. A wait-free implementation of a multi-writer snapshot object has also been described. Finally, the chapter has introduced the notion of an immediate snapshot object and presented both an iterative implementation and a recursive implementation of it.

It is important to insist on the fact that snapshot objects are fundamental objects in crash-prone concurrent environments. This is because they can be used to record the last consistent global state of an application (the value deposited by each process being its last meaningful local state).

8.7 Bibliographic Notes

- Single-writer snapshot objects were introduced independently by Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit [2], and J. Anderson [21].

  The implementation of a single-writer snapshot object described in Fig. 8.6 is due to these authors.

- The best implementation of a single-writer snapshot object known so far requires $O(n \log_2 n)$ accesses to base read/write registers [39]. Finding the corresponding lower bound remains an open problem [93].

- The complexity of update operations of snapshot objects is addressed in [31].

- The extension of the previous implementation to infinitely many processes given in Fig. 8.8 is due to M.K. Aguilera [14].

- Multi-writer snapshot objects were introduced by J. Anderson [22], and M. Inoue, W. Chen, T. Masuzawa and N. Tokura [165].

  Several implementation of multi-writer snapshot objects have been proposed, e.g., [167].

- The implementation of a multi-writer snapshot object described in Fig. 8.10 is due to D. Imbs and M. Raynal [162].

- The notion of partial snapshot was introduced in [36]. A partial snapshot object provides the processes with an operation denoted partial_snapshot($\langle x_1, \ldots, x_y \rangle$), where $\langle x_1, \ldots, x_y \rangle$ denotes the sequence of y components for which the invoking process wants to atomically obtain the values. (If the sequence contains all components, partial_snapshot() boils down to snapshot().) Efficient algorithms that implement partial snapshot objects are described in [158].

- The notion of an immediate snapshot object and its implementation described in Fig. 8.14 are due to E. Borowsky and E. Gafni [53].

  Such objects are central to the definition of iterated computing models [54, 230]. (These distributed computing models are particularly well suited to investigate the computational power of asynchronous systems prone to process crashes.)

- The recursive construction of an immediate snapshot object presented in Sect. 8.5.4 is due to E. Gafni and S. Rajsbaum [112].

8.8 Problem

1. Considering an asynchronous read/write system of n processes where any number of processes may crash, enriched with LL/SC operations (as defined in Sect. 6.3.2), design an efficient partial snapshot implementation of a multi-writer snapshot object.

   Solution in [158].

Chapter 9
Renaming Objects from Read/Write Registers Only

This chapter is devoted to the *renaming* problem. After having presented the problem, it describes three wait-free implementations of one-shot renaming objects. These implementations, which are all based on read/write registers, differ in their design principles, their cost, and the size of the new name space allowed to the processes. Finally, the chapter presents a long-lived renaming object based on registers stronger than read/write registers, namely test&set registers.

Keywords Adaptive algorithm · Immediate snapshot object · Grid of splitters · Recursive algorithm · Renaming object (One-shot versus long-lived).

9.1 Renaming Objects

9.1.1 The Base Renaming Problem

The underlying system The system that is considered is a static system made up of n asynchronous sequential processes $p_1, \ldots, p_n$. Moreover, any number of processes may crash. The processes communicate by accessing atomic read/write registers only. A process *participates* in a run, if it accesses the shared memory at least once in the corresponding run.

Indexes When considering a process p_i, the subscript i is not the identity (name) of process p_i but its *index*. Indexes are used for addressing only; they cannot be used for computing new names (this will be precisely defined below in the statement of the "index independence" property).

Initial names Each process p_i has an initial (permanent) name denoted id_i, which is initially known only by itself. Such a name can be seen as a particular value defined in p_i's initial context that uniquely identifies it (e.g., its IP address). Hence, for any process p_i we have $id_i \in \{1, \ldots, N\}$, where N is the size of the initial name. Moreover,

N is very big when compared to n. As an example, let us consider $n = 100$ processes whose names are made up of eight letters. We have then $N = 8^{26}$ and consequently $n \ll N$.

Initially a process p_i knows only n and id_i. It does not know the initial names id_j of the $n - 1$ other processes p_j. Moreover, two initial names id_i and id_j can only be compared (with $<$, $=$, or $>$).

9.1.2 One-Shot Renaming Object

The aim of such an object is to allow the processes to obtain new names in a new name space whose size M is much smaller than N. Such an object solves the M-renaming problem. To that end, the object offers to the processes a single operation denoted new_name() which can be invoked at most once by each process and returns them values (new names). The object is formally defined by the following properties:

- Liveness. The invocation of new_name() by a correct process terminates.
- Validity. A new name is an integer in the set $[1..M]$.
- Agreement. No two processes obtain the same new name.
- Index independence. $\forall\ i, j$, if a process whose index is i obtains the new name v, that process could have obtained the very same new name v if its index had been j.

The index independence property states that, for any process, the new name obtained by that process is independent of its index. This means that, from an operational point of view, the indexes define only an underlying communication infrastructure, i.e., an addressing mechanism that can be used only to access entries of shared arrays. Indexes cannot be used to compute new names. This property prevents a process p_i from choosing i as its new name without any communication.

9.1.3 Adaptive Implementations

Let p be the number of processes that participate in a renaming execution, i.e., the number of processes that invoke new_name(). Let us observe that the renaming problem cannot be solved when $M < p$. There are two types of adaptive renaming algorithms:

- Size adaptive. An algorithm is size-adaptive if the size M of the new name space depends only on p, the number of participating processes. We have then $M = f(p)$, where $f(p)$ is a function of p such that $f(1) = 1$ and, for $2 \leq p \leq n$, $p - 1 \leq f(p-1) \leq f(p)$. If M depends only on n (the total number of processes), the algorithm is not size-adaptive.

- Time adaptive. An algorithm is time-adaptive if its time complexity (number of accesses to underlying atomic read/write registers) depends only on p. If its time complexity depends only on n, it is not time-adaptive.

Let us consider an execution of a size-adaptive algorithm in which a single process p_i participates. It follows from the definition of "size-adaptive", that that it obtains the new name 1 whatever its index i. Hence, any size-adaptive algorithm has to satisfy the index independence property.

9.1.4 A Fundamental Result

A lower bound on the size of the new name space An important theoretical result associated with the renaming problem in asynchronous read/write systems, due to M. Herlihy and N. Shavit (1999), is the following one. Except for some "exceptional" values of n, the value $M = 2n-1$ is the lower bound on the size of the new name space. For the exceptional values of n, which have been characterized by A. Castañeda and S. Rajsbaum (2008), we have $M = 2n - 2$ (more precisely, there is a $(2n - 2)$-renaming algorithm for the values of n such that the integers in the set $\{\binom{n}{i} \,:\, 1 \le i \le \lfloor \frac{n}{2} \rfloor\}$ are relatively prime).

This means that $M = 2p - 1$ is a lower bound for size-adaptive algorithms (in that case, there is no specific value of p that allows for a lower bound smaller than $2p - 1$). Consequently, the use of an optimal size-adaptive algorithm means that, if "today" p' processes acquire new names, their new names belong to the interval $[1..2p' - 1]$. If "tomorrow" p'' additional processes acquire new names, these processes will have their new names in the interval $[1..2p - 1]$, where $p = p' + p''$.

The price due to the communication by read/write registers only The lower bound $M = 2n-1$ (or $M = 2n-2$) for implementations which are not size-adaptive or $M = 2p-1$ for implementations which are size-adaptive defines the price that has to be paid by any wait-free implementation of the renaming problem when processes communicate by accessing read/write registers only.

This means that, when considering wait-free implementations of optimal size-adaptive M-renaming (i.e., when considering $M = 2p - 1$, where p is the number of participating processes), while only p new names are actually needed, obtaining them requires a space of size $2p-1$ in which $p-1$ new names will never be used and it is impossible to know in advance which of these new names will not be used. This intrinsic uncertainty is the price to pay to obtain wait-free implementations based on read/write atomic registers only.

It follows that, if one wants to design an M-renaming object where $p \le M < 2p - 1$, one has to consider a system with operations which are computationally stronger than simple read/write atomic registers (e.g., test&set(); see Sect. 9.7).

9.1.5 Long-Lived Renaming

In the long-lived renaming problem, a process can repeatedly acquire a new name and then release it. (Long-lived renaming can be useful in systems in which processes acquire and release identical resources.)

So, a long-lived renaming object offers two operations: new_name(), which allows a process to acquire a new name; and release_name(), which allows it to release the new name it has previously acquired.

While nearly all this chapter is devoted to one-shot renaming, its last section presents a simple long-lived renaming object based on test&set registers.

9.2 Non-triviality of the Renaming Problem

The aim of this section is to give an intuition of the difficulty of the renaming problem and its underlying algorithmic principles. To that end we use a simple example.

A simple example with three scenarios Let us consider a system with two asynchronous crash-prone processes p and q that want to acquire new names. They have to coordinate to ensure they do not choose the same new name.

To coordinate, they share two single-writer/multi-reader registers $X[1]$ and $X[2]$. Only p can write into $X[1]$, while only q can write into $X[2]$. Both of them can read $X[1]$ and $X[2]$. Let us consider only one communication exchange, namely p writes to $X[1]$, then reads $X[2]$, and similarly q first writes to $X[2]$, then reads from $X[1]$. Initially processes are identical except for their initial names, so the only useful thing to communicate to the other process is the initial name. There are essentially three scenarios:

- Scenario 1. In this scenario, process p writes (e.g., its initial name) to $X[1]$ to inform q that it wants to acquire a new name, but when p reads the shared register $X[2]$, q has not yet written it (e.g., because it is slow). Hence, p does not "see" that q is competing for a new name. Differently, when q reads $X[1]$, it "sees" that p is competing for a new name.
- Scenario 2. This scenario is the same as the previous one, except that p and q are inverted. Hence, in this scenario, q does not "see" that p is competing for a new name, while p sees that q is competing for a new name.
- Scenario 3. In this symmetric scenario, concurrently p writes into $X[1]$ while q writes into $X[2]$, and then each of them discovers that the other one is competing. Here, each process "sees" that the other one is competing for a new name.

These three possible scenarios are represented in a single graph in Fig. 9.1. Each vertex corresponds to the final state of a process, in one of the scenarios. The scenarios are represented with edges. Two vertices belong to an edge if and only if the corresponding final states appear in the same scenario.

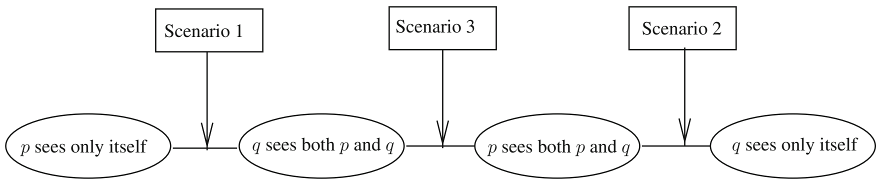


Fig. 9.1 Uncertainties for 2 processes after one communication exchange

The difficulty comes from the fact that in scenario 1, q does not know if p sees it or not. More explicitly, q cannot distinguish scenario 1 and scenario 3. A symmetric situation occurs for p which cannot distinguish scenario 2 and scenario 3; it is in the same state in both; that is, if p is going to choose a new name in this state, it will have to be the same name for both scenario 2 and scenario 3. Hence, we could label the corresponding vertex in the graph with that decision. The graph of Fig. 9.1 represents the global structure of these indistinguishability relations: in this example, a simple path of length 3.

How to address the problem In order to think about the design of an algorithm, let us assume that, whenever a process does not see the other process (because it has crashed or is very slow), it chooses the new name 1. This can be done without loss of generality because the space of initial names is big, hence for every algorithm there exist two processes such that each one of them picks the same new name when it does not see the other; recall that processes are initially identical, except for their initial names, and hence two processes with the same initial name, in this kind of scenario, chose the same name. Consequently, p chooses the new name 1 in scenario 1 and q chooses the new name 1 in scenario 2. Hence, the end vertices of the graph in Fig. 9.1 may be labeled 1.

Let us now look at scenario 3. Process q sees p and is aware that p may have not seen it (this is because q cannot distinguish scenario 1 and scenario 3). To avoid conflict (in case we are in scenario 1 in which case p chooses new name 1), q chooses new name 2. In that case, p (that does not know if the real scenario is scenario 2 or scenario 3) has no choice: it has to choose the new name 3 to ensure that no two processes have the same new name.

This simple observation shows that the renaming problem can be solved for two processes with size of the new name space equal to 3. Let us observe that scenario 4, in which no process sees the other one, cannot happen. This is due to fact that processes communicate by writing and reading a shared memory made up of atomic registers, and each of them writes the shared memory before reading it. The corresponding algorithm is as follows. When p or q wants to acquire a new name, it first deposits its initial name into the shared memory. If it sees only itself, it takes the new name 1. Otherwise, if its initial name is greater than the one of the other process, it takes the new name 2, and if it is smaller, it takes the new name 3. (This simple algorithm will be extended to any number of processes in Sects. 9.5 and 9.6.)

Could it be possible to solve the problem for two processes with two new names only? The previous discussion shows that the answer is "no", if each process is limited to a single communication round (during which it writes and then reads). What if processes are not restricted to one communication round? Perhaps surprisingly, the answer remains "no". This is because the two endpoints of the uncertainty graph (Fig. 9.1) always remain connected. These two endpoints represents the scenario where neither p nor q sees the other process. In these extreme cases, each has to choose the new name 1, and again it would be impossible for p and q to pick only 1 or 2 in the internal nodes, because an edge with equal new names in its endpoints would be unavoidable.

9.3 A Splitter-Based Optimal Time-Adaptive Implementation

This section presents a time-adaptive renaming algorithm that is optimal; namely, when p processes participate, a process executes at most $O(p)$ shared memory accesses. This algorithm is also size-adaptive but not optimal in that respect. The size of its new name space is $M = p(p+1)/2$. This implementation is due to M. Moir and J. Anderson (1995).

Internal representation: a grid of splitters The splitter object was introduced in Sect. 5.2.1. Such an object has a single operation denoted direction() which, (a) returns a value $v \in \{stop, right, down\}$ (the value *down* is used instead of *left* for simplicity), and (b) is such that, if x processes invoke direction(), at most $(x-1)$ processes obtain the value *right*, at most $(x-1)$ processes obtain the value *left*, and at most 1 obtains the value *stop*, and (c) if only one process invokes direction() it obtains the value *stop*. Moreover, an invocation of direction() costs at most four accesses to the atomic MWMR registers used to implement a splitter.

The underlying internal representation is made up of a half-grid of splitters denoted $SP[1..n, 1..n]$. Such a grid for $n = 5$ is depicted in Fig. 9.2.

Underlying principle The idea consists in using a half-grid of $n(n+1)/2$ splitters. Each splitter is assigned a name according to a diagonal numbering: $SP[1, 1]$ has name 1, $SP[2, 1]$ has name 2, $SP[1, 2]$ has name 3, etc. More generally, the name of the splitter $SP[r, d]$ is $(d+r-1)(d+r-2)/2 + r$.

A process p_i first enters the left corner of the grid, i.e., the splitter $SP[1, 1]$ whose name is 1. Then, it moves along the grid according to the values (*down* or *right*) it obtains from the splitters it visits until it arrives at a splitter at which it obtains the value *stop*. Finally, it takes as its new name the name statically assigned to the splitter at which it stops.

The algorithm implementing the operation new_name() The resulting algorithm is described in Fig. 9.3. A process p_i manages two local variables d and r (for going down or right, respectively). It invokes first $SP[1, 1]$.direction() (line 1) and then moves in the grid according to the values it obtains from the splitters it visits (lines

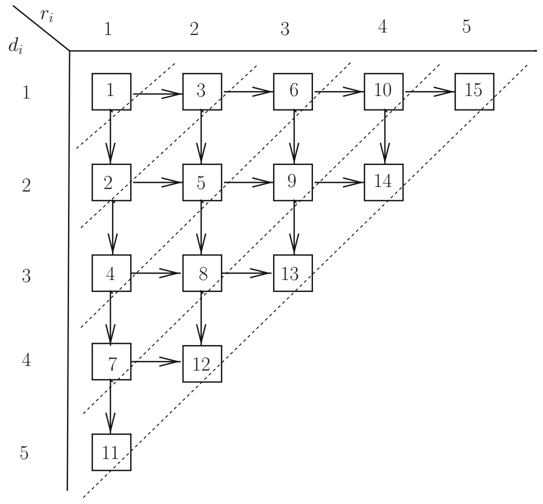


Fig. 9.2 A grid of splitters for renaming

3–6). It follows from the properties of the splitter objects that, if p processes invoke $SP[1, 1]$.direction(), each visits at most p splitters before stopping.

Let us remark that the initial name id_i of p_i is used only to distinguish two processes inside the underlying splitter objects.

Theorem 39 *The implementation described in Fig. 9.3 is a wait-free size-adaptive implementation of an M-renaming object where $M = p(p+1)/2$ and p is the number of participating processes. Moreover, this implementation requires $O(p)$ accesses to underlying atomic read/write shared registers and is consequently time-optimal.*

Proof It follows from the properties of the splitter objects and their grid structure that a process progresses along a path of length at most p before returning the value

```
operation new_name($id_i$) is
(1)   $d \leftarrow 1$; $r \leftarrow 1$; $move \leftarrow down$;
(2)   while ($move \neq stop$) do
(3)       $move \leftarrow SP[d, r]$.direction($id_i$);
(4)       case ($move = right$) then $r \leftarrow r + 1$
(5)            ($move = down$) then $d \leftarrow d + 1$
(6)            ($move = stop$) then exit while loop
(7)       end case
(8)   end while;
(9)   let $res \leftarrow (d + r - 1)(d + r - 2)/2 + r$;
(10)  % $res$ is the number associated with the splitter $SP[d, r]$ %
(11)  return($res$)
end operation.
```

Fig. 9.3 Moir–Anderson grid-based time-adaptive renaming (code for p_i)

stop and stopping. It follows that any invocation of new_name() by a correct process terminates. Moreover, as no more than one process stops at each splitter, and any two splitters have different names, it follows that no two processes can obtain the same new name.

The validity property (the new names belong to the interval $[1..p(p+1)/2]$) follows directly from the fact that the static assignment of names to splitters is a diagonal assignment. The index independence property is trivially satisfied: indexes are used neither in the code of new_name() nor in the code of direction().

Finally, as each invocation of direction() issues at most four accesses to atomic read/write registers and a process travels along a path of at most p splitters, it follows that an invocation of new_name() issues at most $4p$ accesses to atomic registers. Hence, $O(p)$ is an upper bound on the time complexity of an invocation of new_name() (which is clearly optimal for p participating processes as each participating process needs to access the underlying shared memory at least once). □

9.4 A Snapshot-Based Optimal Size-Adaptive Implementation

This section presents a simple size-adaptive M-renaming implementation that provides the participating processes with an optimal new name space, i.e., $M = 2p - 1$, when the processes can cooperate through read/write registers only. This implementation, which is due to H. Attiya and J. Welch (2004), is an adaptation to asynchronous read/write shared memory systems of a message-passing algorithm due to H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk (1990).

9.4.1 A Snapshot-Based Implementation

Internal representation: a snapshot object The internal representation of the renaming object is is a single-writer snapshot object denoted *STATE*. As we have seen this object is an array of SWMR atomic registers denoted $STATE[1..n]$ such that $STATE[i]$ can be written only by p_i while the whole array can be atomically read by p_i by invoking *STATE*.snapshot(). Each atomic register $STATE[i]$ is a pair made up of two fields: $STATE[i].init\_id$, whose aim is to contain the initial name of p_i, while the aim of $STATE[i].prop$ is to contain the last proposal for a new name issued by p_i. Each $STATE[i]$ is initialized to $\langle \perp, \perp \rangle$.

The algorithm implementing the operation new_name() This algorithm is described in Fig. 9.4. The local register $prop_i$ contains p_i's current proposal for a new name. When p_i invokes new_name(id_i), it sets $prop_i$ to 1 (line 1), and enters a **while** loop (lines 2–13) that it will exit after it has obtained a new name [statement return($prop_i$), line 6].

```
operation new_name($id_i$) is
(1)  $prop_i \leftarrow 1$;
(2)  while $true$ do
(3)    $STATE[i] \leftarrow \langle id_i, prop_i \rangle$;
(4)    $state_i \leftarrow STATE$.snapshot();
(5)    if $(\forall\, j \neq i :\ state_i[j].prop \neq prop_i)$
(6)      then return $(prop_i)$
(7)      else let $set1$   = $\{state_i[j].prop \mid (state_i[j].prop \neq \bot) \wedge (1 \leq j \leq n)\}$;
(8)           let $free$   = the increasing sequence $1, 2, \ldots$ from which
                             the integers in $set1$ have been suppressed;
(9)           let $set2$   = $\{state_i[j].init\_id \mid (state_i[j].init\_id \neq \bot) \wedge (1 \leq j \leq n)\}$;
(10)          let $r$      = rank of $id_i$ in $set2$;
(11)          $prop_i$     $\leftarrow$ the $r$th integer in the increasing sequence $free$
(12)   end if
(13) end while
end operation.
```

Fig. 9.4 A simple snapshot-based wait-free size-adaptive $(2p-1)$-renaming implementation (code for p_i)

The principle that underlies the algorithm is the following. A new name can be considered as a slot, and processes compete to acquire free slots in the interval of slots $[1..2p-1]$. After entering the loop, a process p_i first updates $STATE[i]$ (line 3) to announce to all processes its current proposal for a new name (let us note that it also implicitly announces it is competing for a new name).

Then, thanks to the snapshot() operation on the snapshot object *STATE* (line 4), p_i obtains a consistent view (saved in the local array $state_i$) of the system global state. Let us note that this view is consistent because it was obtained from an atomic snapshot operation. Then the behavior of p_i depends on the state of the shared memory it has obtained, more precisely on the value of the predicate

$$\forall\, j \neq i :\ state_i[j].prop \neq prop_i.$$

There are two cases:

- Case 1: the predicate is true. This means that, according to the global state obtained by p_i, no process p_j is competing with p_i for the new name $prop_i$. In that case, p_i considers the current value of $prop_i$ as its new name and consequently returns it (line 6).
- Case 2: the predicate is false. In that case, several processes are competing to obtain the same new name $prop_i$. So, p_i constructs a new proposal for a new name and enters the loop again. This proposal is built by p_i from the global state of the system it has obtained and has saved in $state_i$ (line 4).

The set

$$set1 = \{state_i[j].prop \mid (state_i[j].prop \neq \bot) \wedge (1 \leq j \leq n)\}$$

(line 7) contains the new name proposals (as known by p_i), while the set

$$set2 = \{state_i[j].init\_id \mid (state_i[j].init\_id \neq \bot) \wedge (1 \leq j \leq n)\}$$

(line 9) contains the initial names of the processes that p_i sees as competing for obtaining a new name.

The determination of a new proposal by p_i is based on these two sets: *set*1 is used in order not to propose a new name already proposed, while *set*2 is used to determine a free slot. This determination is done as follows.

First, p_i considers the increasing sequence (denoted *free*) of the integers that are "free" and can consequently be used to define new name proposals. This is the sequence of the increasing positive integers from which the proposals in *set*1 have been suppressed (line 8). Then, p_i computes its rank r among the processes that (from its point of view as given by $state_i[1..n]$) want to acquire a new name (lines 9–10). Finally, given the sequence *free* and r, p_i defines its new name proposal as the rth integer in the sequence *free* (line 11).

9.4.2 Proof of the Implementation

Theorem 40 *The implementation described in Fig. 9.4 is a wait-free optimal size-adaptive implementation of a* M*-renaming object where* $M = 2p - 1$ *and* p *is the number of participating processes.*

Proof Let us first observe that the indexes are used only to address the entries of the snapshot object *STATE*, from which it follows that the implementation satisfies the index independence property.

As far as the renaming agreement property is concerned, let us assume by contradiction that two different processes p_i and p_j obtain the same new name x. Let us assume without loss of generality that the last invocations of *STATE*.snapshot() (line 4) issued by p_i and p_j just before deciding their new names are such that the snapshot invocation of p_i is linearized before the snapshot invocation of p_j.

Let $state_i$ and $state_j$ be the corresponding arrays obtained by p_i and p_j just before returning their new names. It follows (a) from the previous linearization order that $state_j[i] = x$ and (b) from the fact that both return the new name x that we have $state_i[i] = x$ and $state_j[j] = x$. Hence, when evaluated by p_i, the predicate of line 5 is false and consequently p_j cannot return x at line 6. This contradicts the initial assumption and concludes the proof of the agreement property.

As far as the validity property is concerned we have the following. Let us consider a run in which at most p processes participate and let p_i be a process that returns a new name (line 6). If the new name is 1, the validity property is trivially satisfied. Hence, let us consider that the new name of p_i is greater than 1. it follows from the very definition of the value p that, when p_i has defined its last proposal for its new name (line 11), at most $p - 1$ processes have already defined new name proposals. Hence, when considering the pair $(set2, r)$ defined at lines 9 and 10, the rank of id_i in $set2$ is at most p (it is p if id_i is the greatest initial identity among the p participating processes). It then follows from (a) the definition of the sequence *free* (line 8), (b) $r \in \{1, \ldots, p\}$, and (c) the determination of $prop_i$ at line 11 that p_i proposed as a new name a value $\leq p + (p - 1)$, which completes the proof of the validity property.

For the liveness property, let us assume by contradiction that there is a non-empty subset Q of correct participating processes that do not terminate. Let τ be a time after which all faulty participating processes have crashed and all correct participating processes not in Q have terminated. It follows that there is a time $\tau' \geq \tau$ after which all the processes of Q repeatedly invoke $STATE$.snapshot() at line 4 and always obtain the same array of initial names from $STATE[1..n].init\_id$. Consequently, after τ', the processes of Q obtain distinct ranks in $STATE[1..n].init\_id$ (lines 9–10), each process obtaining always the same rank. Moreover, let p_i be the process of Q which has the smallest initial name (id_i) among the processes of Q and r the rank of id_i in the array $STATE[1..n].init\_id$.

As, after τ', all the processes of Q execute repeatedly the lines 7–11, there is a time $\tau'' \geq \tau'$ such that p_i is the only process that proposes $prop_i = z$ as a new name, where z is the rth integer in its sequence *free* (all other processes of Q propose greater names). Hence, when p_i evaluates the predicate $\forall j \neq i : state_i[j] \neq prop_i$ (line 5) after τ'', it finds it is satisfied and consequently returns z as its new name (line 6), which contradicts the initial assumption and completes the proof of the liveness property. $\square$

9.5 Recursive Store-Collect-Based Size-Adaptive Implementation

This section presents a wait-free size-adaptive implementation of a $(2p-1)$-renaming object. This construction, due to S. Rajsbaum and M. Raynal (2011), is based on a sketch of a renaming construction suggested by E. Gafni and S. Rajsbaum (2010). Interestingly the algorithm implementing the operation new_name() is a recursive shared-memory distributed algorithm. This implementation is optimal with respect to the size of the new name space and time-efficient in the sense that its time complexity is $O(n^2)$.

9.5.1 A Recursive Renaming Algorithm

This section extends to any number of processes the simple renaming algorithm for $n = 2$ processes sketched in Sect. 9.2.

Internal representation: atomic SWMR atomic registers The internal representation of the renaming objects is made up of a three-dimensional array of size $n \times (2n-1) \times 2$ denoted $SM[n..1, 1..2n-1, \{up, down\}]$. Each element of this array is a vector of n atomic SWMR registers. Hence, $SM[x,f,d]$ is a vector with n entries, and $SM[x,f,d][i]$ is an atomic register that can be written only by p_i but read by any process p_j. For every 4-tuple $\langle x,f,d,i\rangle$, $SM[x,f,d][i]$ is initialized to $\perp$.

As far as notation is concerned, $up = 1 = \overline{down}$ and $down = -1 = \overline{up}$.

Underlying principle When a process wants to acquire a new name it invokes new_name(x, *first*, *dir*) with $x = n$, $\mathit{first} = 1$, and $\mathit{dir} = up$. The parameter x is the recursion parameter and takes first the value n (main call), then the values $n-1$, $n-2$, etc. until process p_i decides a new name. Its smallest possible value is 1. Hence (cf., Sect. 8.5.4, differently from sequential recursion, where recursion is usually on the size and the structure of the data that is visited, here recursion is on the number of processes.

The value *up* is used to indicate that the concerned processes are renaming "from left to right" (as far as the new names are concerned), while *down* is used to indicate that the concerned processes are renaming "from right to left". More precisely, when p_i invokes new_name(x, f, up), it considers the renaming space $[f..f + 2x - 2]$ to obtain a new name, while it considers the space $[f - (2x - 2)..f]$ if it invokes new_name($x, f, down$). Hence, a process p_i considers initially the renaming space $[1..2n-1]$, and then (as far p_i is concerned) this space will shrink at each recursive call (going up or going down) until p_i obtains a new name.

The algorithm implementing the operation new_name() The algorithm is presented in Fig. 9.5. A process p_i that invokes new_name(x, *first*, *dir*) first writes its initial name id_i in $SM[x, \mathit{first}, \mathit{dir}][i]$ (line 1) and then reads asynchronously each entry of the array of atomic registers $SM[x, \mathit{first}, \mathit{dir}][1..n]$ (line 2). This array is used to allow the processes that invoke new_name(x, *first*, *dir*) to "coordinate" to obtain new names. More precisely, all processes that compete for new names in $[\mathit{first}..\mathit{first} + 2x - 2]$ if $\mathit{dir} = up$, or $[\mathit{first} - (2x - 2)..\mathit{first}]$ if $\mathit{dir} = down$, deposit their initial names in $SM[x, \mathit{first}, \mathit{dir}]$ (line 1). Then, according to the value (saved in the local array $\mathit{competing}_i$) that a process has read (not atomically) from the vector $SM[x, \mathit{first}, \mathit{dir}][1..n]$ (line 2), the behavior of the set X of processes that invoke new_name(x, *first*, *dir*) is similar to the behavior of splitter (see below).

For each triple (x, f, d), all invocations new_name(x, f, d) coordinate their respective behavior with the help of the size n array of atomic registers $SM[x, f, d]$ $[1..n]$. The local variable $\mathit{competing}_i$ is an array of n local variables such that $\mathit{competing}_i[j]$ contains either $\perp$ or id_j, the initial name of p_j (line 1).

The following notations are used: $|\mathit{competing}_i|$ denotes the number of entries that are different from $\perp$, while $\max(\mathit{competing}_i)$ is the greatest initial name it contains. As a process p_i deposits its initial name in $SM[x, \mathit{first}, \mathit{dir}][i]$ before reading $SM[x, \mathit{first}, \mathit{dir}][1..n]$, it follows that $\mathit{competing}_i$ contains at least one non-$\perp$ entry when it is read by p_i.

```
operation new_name($x$, $first$, $dir$) is
        % $x$ ($n \geq x \geq 1$) is the recursion parameter %
(1)     $SM[x, first, dir]$.store($id_i$);
(2)     $competing_i \leftarrow SM[x, first, dir]$.collect();
(3)     if $|competing_i| = x$
(4)       then $last \leftarrow first + dir(2x-2)$;
(5)            if $id_i = \max(competing_i)$
(6)              then $res_i \leftarrow last$
(7)              else $res_i \leftarrow$ new_name($x-1$, $last + \overline{dir}$, $\overline{dir}$)
(8)            end if
(9)       else $res_i \leftarrow$ new_name($x-1$, $first$, $dir$)
(10)    end if;
(11)    return($res_i$)
end operation.
```

Fig. 9.5 Recursive optimal size-adaptive renaming (code for p_i)

Let us observe that, if only p processes invoke new_name(n, 1, up), $p < n$, then all of them will invoke the algorithm recursively, first with new_name($n-1$, 1, up), then new_name($n-2$, 1, up), etc., until the call new_name(p, 1, up). Only at this point, when p processes invoke new_name(p, 1, up), does the behavior of a participating process p_i depend on the concurrency pattern (namely, it may or may not invoke the algorithm recursively, and with either up or $down$).

Splitter-like behavior associated with $SM[x, first, dir]$ Considering the (at most) x processes that invoke new_name(x, $first$, dir), the "splitter" behavior (adapted to renaming) associated with $SM[x, first, dir]$ is defined by the following properties. Let $x' = x-1$.

- At most $x' = x-1$ processes invoke new_name($x-1$, $first$, dir) (line 9). Hence, these processes will obtain new names in an interval of size $(2x'-1)$ as follows:
  - If $dir = up$, the new names will be in the "going up" interval $[first..first + (2x'-2)]$,
  - If $dir = down$, the new names will be in the "going down" interval $[first - (2x'-2)..first]$.
- At most $x' = x-1$ processes invoke new_name($x-1$, $last + \overline{dir}$, $\overline{dir}$) (line 7), where $last = first + dir(2x-2)$ (line 4). Hence, these $x' = x-1$ processes will obtain their new names in a renaming space of size $(2x'-1)$ starting at $last+1$ and going from left to right if $\overline{dir} = up$, or starting at $last-1$ and going from right to left if $\overline{dir} = down$. Let us observe that the value $last \pm 1$ is considered as the starting name because the slot $last$ is reserved for the new name of the process (if any) that stops during its invocation of new_name(x, $first$, dir) (see next item).
- At most one process "stops", i.e., defines its new name as $last = first + dir(2x-2)$ (lines 4 and 6). Let us observe that the only process p_k that can stop is the one such that id_k has the greatest value in the array $SM[x, first, dir][1..n]$ (line 5), which contains then exactly x old names (line 3).

9.5.2 An Example

This section presents an example of an execution of the renaming object described in Fig. 9.5. It considers four processes p_1, p_2, p_3, and p_4. Their initial names are such that $id_1, id_2 < id_4 < id_3$.

First process p_3 executes alone Process p_3 invokes new_name(4, 1, *up*)() while (for the moment) no other process invokes the renaming operation. It follows from the algorithm that p_3 invokes recursively new_name(3, 1, *up*)(), then new_name (2, 1, *up*)(), and finally new_name(1, 1, *up*)(). During the last invocation, it obtains the new name 1. This is illustrated in Fig. 9.6. As, during its execution, p_3 sees only $p = 1$ process (namely, itself), it decides consistently in the new name space $[1..2p - 1] = 1$.

Then processes p_1 and p_4 invoke new_name() After p_3 has obtained a new name, both p_1 and p_4 invoke new_name(4, 1, *up*)() (See Fig. 9.7). As they see only three processes that have written their old names into $SM[4, 1, up]$, both concurrently invoke new_name(3, 1, *up*)() and consequently both compute $last = 1 + (2 * 3 - 2) = 5$. Hence their new name space is [1..5].

Now, let us assume that p_1 stops executing during some time and p_4 executes alone. As it has not the greatest initial names among the processes that have accessed $SM[3, 1, up]$ (namely, p_1, p_3 and p_4), p_4 invokes first new_name(2, 4, *down*)() and then recursively new_name(1, 4, *down*)() and finally obtains the new name 4.

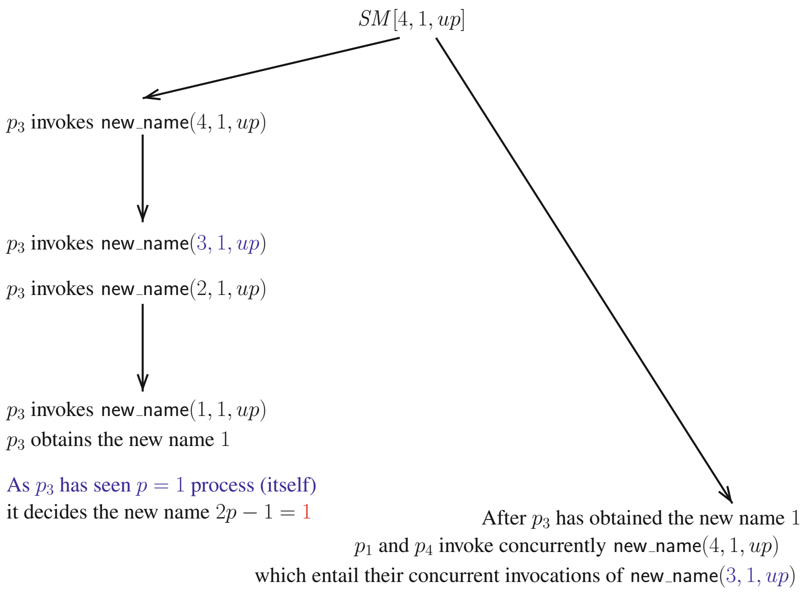

$SM[4, 1, up]$
p_3 invokes new_name$(4, 1, up)$
p_3 invokes new_name$(3, 1, up)$
p_3 invokes new_name$(2, 1, up)$
p_3 invokes new_name$(1, 1, up)$
p_3 obtains the new name 1
As p_3 has seen $p = 1$ process (itself)
it decides the new name $2p - 1 = 1$
After p_3 has obtained the new name 1
p_1 and p_4 invoke concurrently new_name$(4, 1, up)$
which entail their concurrent invocations of new_name$(3, 1, up)$

Fig. 9.6 Recursive renaming: first, p_3 executes alone

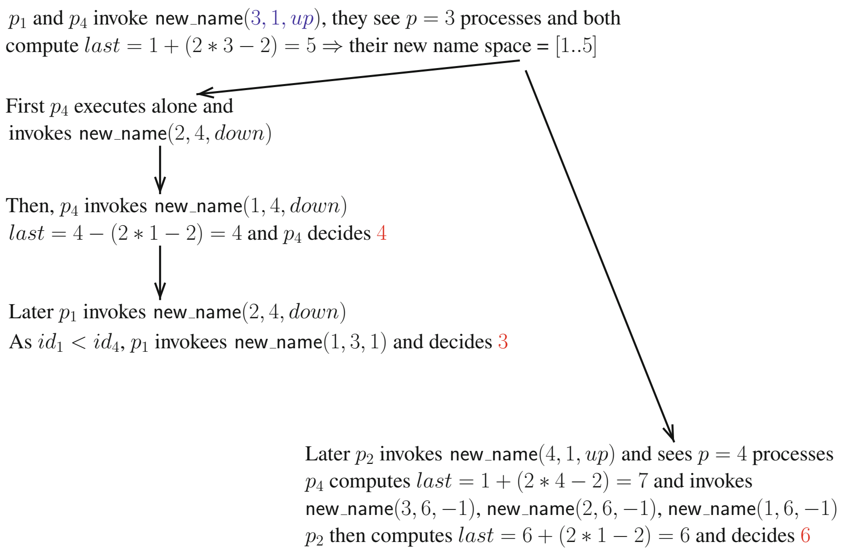

p_1 and p_4 invoke new_name$(3, 1, up)$, they see $p = 3$ processes and both compute $last = 1 + (2 * 3 - 2) = 5 \Rightarrow$ their new name space = $[1..5]$
First p_4 executes alone and invokes new_name$(2, 4, down)$
Then, p_4 invokes new_name$(1, 4, down)$ $last = 4 - (2 * 1 - 2) = 4$ and p_4 decides 4
Later p_1 invokes new_name$(2, 4, down)$ As $id_1 < id_4$, p_1 invokees new_name$(1, 3, 1)$ and decides 3
Later p_2 invokes new_name$(4, 1, up)$ and sees $p = 4$ processes p_4 computes $last = 1 + (2 * 4 - 2) = 7$ and invokes new_name$(3, 6, -1)$, new_name$(2, 6, -1)$, new_name$(1, 6, -1)$ p_2 then computes $last = 6 + (2 * 1 - 2) = 6$ and decides 6

Fig. 9.7 Recursive renaming: p_1 and p_4 invoke new_name(4, 1, *up*)()

After p_4 has obtained its new name, p_1 continues its execution and invokes new_name(2, 4, *down*)() and computes $last = 4 - (2 \times 2 - 2) = 2$. As, among the two processes (p_1 and p_4) that access $SM[2, 4, down]$, p_1 does not have the greatest initial name ($id_1 < id_4$), it invokes new_name(1, 3, *up*)() and obtains new name 3.

Let us observe that the new name space attributed to the $p = 3$ processes p_1, p_3, and p_4 (the only ones that, up to now, have invoked new_name(4, 1, *up*)()) is $[1..2p - 1] = [1..5]$.

Finally process p_2 invokes new_name() Let us now assume that p_2 eventually invokes new_name(4, 1, *up*). It sees that $p = 4$ processes have accessed $SM[4, , up]$, and computes $last = 1 + (2 \times 4 - 2) = 7$. The size of the new name space becomes consequently $[1..2p - 1] = [1..7]$.

As it dies not have the greatest initial name among the four processes, p_2 invokes new_name(3, 6, *down*), and then invokes recursively new_name(2, 6, *down*) and new_name(1, 6, *down*) and obtains 6 as its new name.

9.5.3 Proof of the Renaming Implementation

This section shows that the recursive algorithm described in Fig. 9.5 is correct, i.e., that all correct participating processes obtain a new name in the interval $[1..2p - 1]$ (where p is the number of participating processes), and no two new names are identical. Moreover, the process indexes are used only as an addressing mechanism (index independence).

Notation In the following the sentence “process p_i stops in $SM[x,f,d]$” means that p_i executes line 6 during its invocation new_name(x,f,d).

Remark The proof is based on reasoning by induction. This is a direct consequence of the recursive formulation of the algorithm. In that sense the proof provides us with deeper insight into the way the algorithm works.

Lemma 21 *Let $\langle x,f,d\rangle$ be a triple such that $x \geq 1$ and assume that at most x processes invoke the operation* new_name(x,f,d). *At most one process stops in $SM[x,f,d]$ (line 6), at most $(x-1)$ processes invoke* new_name($x-1,f,d$) *(line 9), and at most $(x-1)$ processes invoke* new_name($x-1,f',\overline{d}$) *(line 7).*

Proof Let $u \leq x$ be the number of processes that invoke new_name(x,f,d). If $u < x$, it follows that the predicate at line 3 is false for these processes, which consequently proceed to line 9 and invoke new_name($x-1,f,d$). As $u \leq x-1$, the lemma follows. Let us now consider the case where x processes invoke new_name(x,f,d). We have then the following:

- Let y be the number of processes for which the predicate $|competing_j| = x$ (line 3) is false when they invoke new_name(x,f,d). We have $0 \leq y < x$. It follows from the text of the algorithm that these y processes invoke new_name($x-1,f,d$) at line 9. As $y < x$, the lemma follows for these invocations.
- Let z be the number of processes for which the predicate $|competing_j| = x$ (line 3) is true when they invoke new_name(x,f,d). We have $1 \leq z \leq x$ and $y + z = x$.

  If one of these z processes p_k is such that the predicate of line 5 is true (i.e., $id_k = \max(SM[x,f,d])$), then p_k executes line 6 and stops inside $SM[x,f,d]$. Let us also note that this is always the case if $x = 1$. If $x > 1$, it follows from $y + z = x$ and $y \geq 0$ that $z - 1 \leq x - 1$. Then, the other $z - 1$ processes invoke new_name($x-1,f',\overline{d}$). As $z-1 \leq x-1$, the lemma follows for these invocations.

  If the test of line 5 is false for each of the z processes, it follows that the process p_k that has the greatest old name among the x processes that invoke new_name(x,f,d) is necessarily one of the y previous processes. Hence, in that case, we have $y \geq 1$. As $y + z = x$, this implies $z < x$. It then follows that at most $z \leq x - 1$ processes invoke new_name($x-1,f',\overline{d}$), which concludes the proof of the lemma. □

Lemma 22 *Every correct participant decides a new name.*

Proof As there are at most n participating processes, and each starts by invoking new_name(n, 1, 1), it follows from Lemma 21 that every correct process stops in some $SM[x,*,*]$ for $1 \leq x \leq n$. It then decides a new name at line 6, which proves the lemma. □

Lemma 23 *Let p be the number of participating processes. We have $M = [1..2p-1]$. Moreover, for any pair of participating processes p_i and p_j, we have $res_i \neq res_j$.*

Proof The lemma is trivially true for $p = 1$ (the single process that invokes new_name(n, 1, 1) obtains the new name 1, if it does not crash). A simple case analysis shows that the new names for $p = 2$ are a pair in [1..3] (each pair is actually associated with a set of concurrency and failure patterns).

The rest of the proof is by induction on the number of participating processes. The previous paragraph has established the base cases. Assuming that the lemma is true for all $p' \leq p$ (induction assumption), let us show that it is true for $p+1$ participating processes.

Each of the $p+1$ processes invokes new_name(n, 1, 1). Each of these invocations entails the invocation of new_name($n-1$, 1, 1) (line 9), etc., until the invocation new_name($p+1, f, d$) with $f = 1$ and $d = 1$.

- Let Y be the set of processes p_j (with $|Y| = y$) such that the predicate $|contending_j| = p+1$ (line 3) is false. We have $0 \leq y < p+1$. These processes invoke new_name(p, f, d), etc., until new_name(y, f, d), and due to the induction assumption, they rename (with distinct new names) in $[f..f+2y-2]$, namely $[1..2y-1]$, since $f = d = 1$.
- Let Z be the set of processes p_j (with $|Z| = z$) such that the predicate $|contending_j| = p+1$ (line 3) is true. We have $1 \leq z \leq p+1$ and $y+z = p+1$. At line 4, each of these z processes obtains $last = f + 2(p+1) - 2 = f + 2p$, i.e., $last = 2p+1$ (as $f = 1$).
  - If one of these z processes p_k is such that $id_k = \max(SM[p+1, f, d])$ (line 5), it stops at $SM[p+1, f, d]$), obtaining the name $res = last = f + 2p = 2p+1$ (as $f = 1$).
  - If no process stops at $SM[p+1, f, d]$, we have $1 \leq z \leq p$ and $1 \leq y$ (this is because the process with the greatest old name is then necessarily a process of Y).

  Hence, the $z' = z \leq p$ or $z' = z - 1 \leq p$ processes that do not stop at $SM[p+1, f, d]$ invoke new_name(p, $last - 1, \overline{d}$) (where $f = d = 1$), etc., until new_name($z', f + 2p - 1, \overline{d}$). Due to the induction assumption, these z' processes rename (with distinct new names) in the interval $[(f + 2p - 1) - (2z' - 2)..f + 2p - 1] = [2p - (2z' - 2)..2p]$.
- Hence, when considering the $y + z = p + 1$ processes of $Y \cup Z$, the y processes of Y rename with distinct new names in $[1..2y-1]$ (where $0 \leq y < p+1$), the z' processes of Z rename with distinct names $[2p - (2z' - 2)..2p]$ (where $z' \leq z$ and $y + z = p+1$), and if $z' + 1 = z$, the remaining process of Z obtains the new name $2p+1$. The new name space for the whole set processes $Y \cup Z$ is consequently $[1..2p+1]$.
- It remains to show that a process of Y and a process of Z cannot obtain the same new name. To that end we have to show that the upper bound $2y - 1$ of the new

names of the processes of Y is smaller than the lower bound $2p - (2z' - 2)$ of the new names of the processes of Z, namely $2y - 1 < 2p - (2z' - 2)$, i.e., $2(y + z') < 2(p + 1) + 1$, which is true because $z' \leq z$ and $y + z = p + 1$, which concludes the proof of the lemma. □

Theorem 41 *The algorithm described in Fig. 9.5 is a a size-adaptive* $(2p - 1)$-*renaming algorithm (where p is the number of participating processes). Its step complexity is* $O(n^2)$.

Proof The fact that no two new names are identical and that the new name space is $[1..2p - 1]$ is proved in Lemma 23. The fact that any correct participating process decides a new name is proved in Lemma 22. Finally the index independence property follows directly from the text of the algorithm: the process indexes are used only in lines 1 and 2 where they are used to address entries of arrays.

It is easy to see that the step complexity is $O(n^2)$. This follows from the fact that writing id_i at line 1 costs one shared memory access, reading $SM[x, f, d][1..n]$ costs n shared memory accesses, and a process executes at most n recursive calls. □

9.6 Variant of the Previous Recursion-Based Renaming Algorithm

This section shows how "useless" recursive invocations of the previous algorithms can be saved with the help of immediate snapshot objects (which replace the store-collect objects).

9.6.1 A Renaming Implementation Based on Immediate Snapshot Objects

Eliminate recursive calls Let us consider the case where $y < n$ processes participate concurrently in the previous size-adaptive implementation described in Fig. 9.5. It is easy to see that these processes invoke at line 9 first new_name(n, 1, 1) and then recursively new_name($n - 1$, 1, 1), new_name($n - 2$, 1, 1), etc., until new_name(y, 1, 1). It is only from this invocation that the processes start doing "interesting" work. Hence, the question: Is it possible to eliminate (whatever the value of y) these useless invocations?

Solving this issue amounts to directing a process to "jump" directly to the invocation new_name(y, 1, 1) such that we have $|competing_i| = y$ in order to execute only lines 4–8 of the algorithm of Fig. 9.5. Interestingly, the property we are looking for is exactly what is provided by the update_snapshot() operation of the one-shot immediate snapshot object defined in Sect. 8.5. This operation directs a process to the appropriate *concurrency level* as defined by the number of processes that the invoking process perceives as executing concurrently with it.

The algorithm implementing the operation new_name() The resulting unimplementation based on immediate snapshot objects is described in Fig. 9.8. Interestingly, this algorithm was proposed by E. Borowsky and E. Gafni (1993).

A process p_i invokes new_name(ℓ, *first*, *dir*), where *first* $=$ *dir* $= 1$ and ℓ is a list initialized to $\langle n\rangle$. This list is the recursion parameter. The lists generated by the recursive invocations of all participating processes define a tree that is the recursion tree associated with the whole execution. The list $\langle n\rangle$ is associated with the root of this tree. These lists are used to address the appropriate entry of the array of one-shot immediate snapshot objects SM.

Similarly to the implementation described in Fig. 9.5 where each $SM[x, f, d]$ is a store-collect object, each $SM[\ell]$ is now a one-shot immediate snapshot object implemented by an array of n SWMR atomic registers (such that only p_i can write $SM[\ell][i]$). Moreover, at most s processes invoke the operation $SM[\ell]$.update_snapshot(), where s is the integer which is the last element of the list ℓ. More generally, the elements of the list indicate the current recursion path.

A process p_i first invokes $SM[\ell]$.update_snapshot(id_i) (line 1). This allows it to access its concurrency level thereby skipping all useless recursive invocations. It then executes only "useful work" (lines 2–7). When considering these lines, the only difference with respect to the algorithm of Fig. 9.5 lies in the management of the recursion parameter needed in the case where $id_i \neq \max(competing_i)$. The value of the recursion parameter (which is now an extended list) used at line 6 is defined from the current recursion parameter (the list $\ell = \langle n_1, n_2, \ldots, n_\alpha\rangle$ where $n_1 = n$) and the size of the actual concurrency set $competing_i$. The new list is $next\ell = \langle n_1, n_2, \ldots, n_\alpha, |competing_i|\rangle$. This value is computed at line 5 where $\oplus$ is used to denote concatenation.

Let us observe that the recursive invocations entailed by new_name($\langle n\rangle$, 1, 1) issued by a process p_i are such that $n_1 = n > n_2 > .. > n_\alpha > |competing_i| > 0$ (from which it is easy to prove that any invocation new_name($\langle n\rangle$, 1, 1) always terminates).

```
operation new_name($\ell$, $first$, $dir$) is
      % the increasing list $\ell$ is the recursion parameter %
(1)   $competing_i \leftarrow SM[\ell]$.update_snapshot($id_i$);
(2)   $last \leftarrow first + dir(2 \times |competing_i| - 2)$;
(3)   if $id_i = \max(competing_i)$
(4)      then $res_i \leftarrow last$
(5)      else $next\ell \leftarrow \ell \oplus |competing|$;
(6)           $res_i \leftarrow$ new_name($next\ell$, $last + \overline{dir}$, $\overline{dir}$)
(7)   end if;
(8)   return($res_i$)
end operation.
```

Fig. 9.8 Borowsky and Gafni's recursive size-adaptive renaming algorithm (code for p_i)

Number of shared memory accesses As far as the step complexity (measured as the number of shared memory accesses) is concerned, we have the following. Each invocation of $SM[\ell]$.update_snapshot() involves n shared memory accesses and a process issues at most n recursive invocations. It follows that the number of accesses to atomic SWMR atomic registers is upper–bounded by $O(n^2)$.

9.6.2 An Example of a Renaming Execution

Let us consider an execution in which $p = 5$ processes participate. Hence, the new name space is $[1..2p - 1] = [1..9]$. To simplify the presentation and without loss of generality we consider the participating processes are $p_1,\ldots,p_5$ and that $old\_name_i = i$. Moreover, let us assume that none of these processes crashes.

At the beginning only p_4 and p_5 are concurrently participating, each invoking new_name($\langle n\rangle$, 1, 1) (the other processes p_1, p_2, and p_3 will invoke new_name($\langle n\rangle$, 1, 1) later). It follows that each of p_4 and p_5 invokes $SM[\langle n\rangle]$.update_snapshot(), which returns to each of them the set $\{id_4, id_5\} = \{4, 5\}$; which they save in $competing_4$ and $competing_5$, respectively. Consequently, each of p_4 and p_5 considers that it is competing for a new name with the other process (line 1). Consequently, as $|competing_4| = |competing_5| = 2$, both p_4 and p_5 compute $last = 1+(2\times 2-2) = 3$ (line 2) and both "reserve" the new name $last = 3$ for the one of them with the greatest initial name. Hence, p_5 executes the lines 3–4 and receives the new name 3.

In contrast, p_4 executes lines 5–6 and invokes new_name($\langle n, 2\rangle, 2, -1$). This invocation by p_4 entails the invocation $SM[\langle n, 2\rangle]$.update_snapshot(id_4), which returns it the singleton set $\{id_4\} = \{4\}$ (line 1). Consequently, p_4 is such that $last = 2 - (2 \times 1 - 2) = 2$ (line 2), and as $\{id_4\} = \max\{4\})$, p_4 obtains the new name $last = 2$.

It follows that, as they were early with respect to the other processes, p_4 and p_5 have obtained the new names 2 and 3, respectively. Let us observe that, if they were the only participating processes, the new name space would be $[1..2p - 1] = [1..3]$. This is due to the fact that the algorithm is size-adaptive.

Moreover, when we consider the tree associated with the execution of all participating processes (this tree is described in Fig. 9.9), p_5 stopped at the root (whose label is $\langle n\rangle$) while p_4 stopped at its descendant labeled $\langle n, 2\rangle$.

Let us now consider that, after p_4 and p_5 have returned from $SM[\langle n\rangle]$.update_snapshot(n), p_1, p_2, and p_3 concurrently invoke new_name($\langle n\rangle$, 1, 1). Their concurrent invocations of $SM[\langle n\rangle]$.update_snapshot(n) return the set $\{1, 2, 3, 4, 5\}$ (the set of the five old names) to each of them (line 2). Hence, they all compute $last = 1 + (2 \times 5 - 2) = 9$ and reserve the new name 9 for the one of them that has the greatest old name, i.e., p_5. (This reservation is just in case p_5 would be competing with them; let us observe that p_5 has already obtained its new name 3 but this is known neither by p_1 nor p_2 nor p_3).

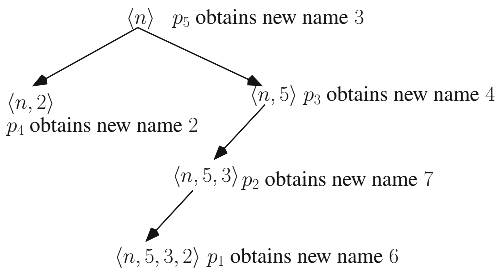

$\langle n\rangle$ p_5 obtains new name 3
$\langle n,2\rangle$
p_4 obtains new name 2
$\langle n,5\rangle$ p_3 obtains new name 4
$\langle n,5,3\rangle$ p_2 obtains new name 7
$\langle n,5,3,2\rangle$ p_1 obtains new name 6

Fig. 9.9 Tree associated with a concurrent renaming execution

It follows that each of p_1, p_2 and p_3 invokes new_name($\langle n, 5\rangle, 8, -1$) (where $8 = last + \overline{dir}$ with $dir = 1$). These vocations give rise to their concurrent invocations of $SM[\langle n, 5\rangle]$.update_snapshot(5), which return to each of them the set $\{1, 2, 3\}$. The three of them compute $last = 8 - (2 \times 3 - 2) = 4$ and reserve the new name 4 for the one of them with the greatest old name, namely p_3. Hence, p_3 obtains new name 4 (lines 3–4).

In contrast, p_2, and p_3 invoke new_name($\langle n, 5, 3\rangle, 5, 1$), where $5 = last + \overline{dir}$ with $dir = -1$ (line 6). Let us consider that both p_2 and p_3 invoke then concurrently $SM[\langle n, 5, 3\rangle]$.update_snapshot(3) from which they both obtain the set $\{2, 3\}$. Then, it is easy to see that p_3 obtains the new name $last = 5 + (2 \times 2 - 2) = 7$ (line 4). Finally p_2 invokes new_name($\langle n, 5, 3, 2\rangle, 6, -1$) and obtains the new name $last = 6 - (2 \times 1 - 2) = 6$.

Let us finally observe that, when a single process participates, whatever its initial name, it obtains the new name 1 from new_name($\langle n\rangle, 1, 1$) when it is at the root of the execution tree.

9.7 Long-Lived Perfect Renaming Based on Test&Set Registers

9.7.1 Perfect Adaptive Renaming

***p*-Renaming** This section presents a simple adaptive long-lived renaming based on atomic test&set registers. Due to their computability power, which is stronger than that of simple atomic read/write registers, test&set registers allow for a new name space with the smallest possible size, where "smallest" means that, due to the very definition of the problem, it is not possible to have a smaller name space whatever the computability power the processes are provided with.

More explicitly, the size of the new name space is $M = p$, where p is the number of participating processes. This is called the *perfect* (size-adaptive) renaming problem.

Anonymous processes In the implementation presented below, the processes are not required to have initial names. They can be fully anonymous. However, the processes have to know n (the total number of processes).

9.7.2 Perfect Long-Lived Test&Set-Based Renaming

Reminder: test&set registers Atomic test&set registers have already been used in previous chapters. Let us remember that such a register TS can take two values: 1 (winner value) and 0 (loser value). It can be accessed by two operations denoted TS.test&set() and TS.reset(), whose sequential sequential specification is as follows. An invocation of TS.test&set() returns the current TS and, whatever its previous value, sets it to 0 (loser value). An invocation of TS.reset() writes 1 into TS.

Internal representation The perfect long-lived renaming object is made up of an array of n test&set registers denoted $TS[1..n]$.

Algorithms for the operations new_name() **and** release_name() The algorithms implementing these operations are described in Fig. 9.10. It is assumed that a process invokes release_name(x) only after it has obtained (and not yet released) the new name x.

When it invokes new_name(), a process executes a **for** loop (lines 1–4) until it has obtained a new name. It first checks if the new name $y = 1$ is available. If $TS[1]$.test&set() returns 1, the new name $y = 1$ is available and the process returns it (lines 2–3). Otherwise, the process process to check if the new name $y = 2$ is available, etc., until the new name $y = n$.

After it has obtained the new name x, a process can release it by invoking release_name(x), which resets the test&set register $TS[x]$ to the value 1 (line 5).

Theorem 42 *The algorithms described in Fig. 9.10 defines a wait-free implementation of a perfect long-lived renaming object.*

```
operation new_name() is
(1)  for $y$ from $1$ to $n$ do
(2)     $r \leftarrow TS[y]$.test&set();
(3)     if $(r = 1)$ then return($y$) end if
(4)  end for
end operation.

operation release_name($x$) is
(5)  $TS[x]$.reset();
(6)  return()
end operation.
```

Fig. 9.10 Simple perfect long-lived test&set-based renaming

Proof Let us first observe that, if p processes are participating, it follows from the fact that the base test&set registers are checked sequentially, first $TS[1]$, then $TS[2]$, etc., that at most the registers $TS[1]$ until $TS[p]$ are accessed by the p processes. It follows from this observation that the implementation is such that the new name space is at most $[1..p]$. (It can be $[1..p']$, where $p' < p$, if new names which have been released are later used by other processes.)

As there is no index, the implementation trivially satisfies the index independence property. The agreement property follows from the fact that, at any time, any test&set register $TS[y]$ has at most one winner (the process which obtained from it the value 1 and has not yet invoked $TS[y]$.reset()).

The liveness (wait-freedom) property of the operation relase_name() follows trivially from its code. For the operation new_name(), it follows from (a) the fact that there are at most n participating processes and (b) that the processes are assumed to release their previous new names before trying to acquire new ones, that when a process invokes new_name(), there is at least one test&set register $TS[y]$ whose value is 1 (winner value). □

9.8 Summary

The focus of this chapter was on the renaming problem. After having defined the problem, the chapter has presented several wait-free implementations of renaming objects in the base asynchronous read/write system. These implementations differ in their time complexity and the size of the new name space granted to processes. A perfect long-lived renaming object based on registers stronger than base read/write registers, namely test&set registers, has also been presented.

9.9 Bibliographic Notes

- The renaming problem was first introduced in the context of asynchronous message-passing systems where processes may crash by H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk in 1990 [29]. It was the first non-trivial problem known to be solvable in asynchronous systems despite process failures.
- An introductory survey of the renaming problem and its connection with distributed computability recently appeared in [66].
- The renaming problem has received a lot of attention in asynchronous systems where processes communicate by reading and writing atomic registers. Size-adaptive or time-adaptive renaming algorithms more involved than the ones that have been presented have been designed. For example a size-optimal algorithm (i.e., $M = 2p - 1$) whose step complexity is $O(n^2)$ is presented in [9], and a

new base object called a *reflector* from which an optimal size-adaptive renaming algorithm is built is introduced in [33].

The lower bounds $M = 2n - 1$, and $M = 2n - 2$ for an infinite number of exceptional values of n, are due to M. Herlihy and N. Shavit [145] and A. Castañeda and S. Rajsbaum [65], respectively.

- The splitter-based renaming algorithm described in Fig. 9.3 is due to M. Moir and J. Anderson [209]. An assertional proof of this time-adaptive renaming algorithm can be found in that paper.
- The notion of long-lived renaming is due to M. Moir and J. Anderson [208, 209].

  The algorithm described in [32], which is due to H. Attiya and A. Fouren, is both size- and time-adaptive for the long-lived renaming problem. Let p be the number of processes that are currently participating (those that have invoked new_name() and not yet invoked release_name()). The algorithm is such that $M = 2p - 1$ and its step complexity is $O(p^4)$.
- The size-adaptive renaming implementation based on atomic read/write registers described in Fig. 9.4 is due to H. Attiya and J. Welch [41]. It is an adaptation to the shared memory context of a message-passing implementation defined in [29]. It is shown in [103] that this algorithm has runs in which invocations of new_name() can give rise to an exponential number of shared memory accesses.
- The recursive size-adaptive renaming implementation based on store-collect objects described in Fig. 9.5 is due to S. Rajsbaum and M. Raynal [229]. It is inspired from a renaming implementation sketch described in [112] (this paper is focused on recursion in distributed algorithms).
- The recursive size-adaptive renaming implementation based on immediate snapshot objects described in Fig. 9.8 is due to E. Borowsky and E. Gafni [53].
- A generalization of the renaming problem for groups of processes is proposed in [106] and investigated in [7]. In this variant, each process belongs to a group and knows the original name of its group. Each process has to choose a new name for its group in such a way that two processes belonging to distinct groups choose distinct new names.
- The relations between renaming objects and other objects which are central to distributed computability such as k-set-agreement [71] have received a lot of attention (e.g., [18, 19, 107, 111, 113, 154, 160, 215, 216] to cite a few).

9.10 Exercises and Problems

1. Considering an anonymous system (i.e., a system in which processes have no initial name) with an unknown number of processes and atomic registers that can

be accessed by the operation test&set(), design an algorithm that allows each process to obtain a unique name.

Solution in [14].

2. Let us consider a system of n processes that have access to one-shot size-adaptive renaming objects and atomic registers. Design an algorithm that implements the operation test&set().

3. Design a long-lived renaming algorithm from read/write atomic registers only.

   Solution in [41].

4. Prove that the adaptive $(2p-1)$-renaming algorithm due to Borowsky and Gafni (described in Sect. 9.6) is correct.

5. A k-set agreement object is a one-shot object that provides processes with a single operation, denoted propose(). This operation allows the invoking process to propose a value v which is passed as an input parameter. It returns to the invoking process a value called *decided value*. The object is defined by the following properties:

   - Liveness. An invocation of propose() by a correct process terminates.
   - Validity. A decided value is a proposed value.
   - Agreement. At most k different values are decided.
   - Integrity. A process decides at most one value.

   Design a size-adaptive M-renaming object, such that $M = p + k - 1$, from any number of atomic read/write registers and k-set agreement objects.

   Solution in [107].

6. A one-shot k-test&set object generalizes a test&set object (which corresponds to a 1-test&set object).

   Such an object provides the processes with an operation denoted test&set$_k$() such that at least one and at most k processes obtain the value 1 (they are the winners) and the other processes obtain the value 0 (they are the losers).

   Design a size-adaptive M-renaming object, where $M = 2p - \lceil \frac{p}{k} \rceil$, from any number of atomic read/write registers and one-shot k-test&set objects.

   Solution in [215].

Part IV
The Transactional Memory Approach

This part of the book is made up of a single chapter devoted to a new approach for designing multiprocess programs, namely the software transactional memory (STM) approach. The idea that underlies this approach is to free programmers from “implementation details” associated with synchronization.

Chapter 10
Transactional Memory

This chapter is devoted to software transactional memories (STM). This concept was first proposed by M. Herlihy and J. Moss (1993), and later refined by N. Shavit and D. Touitou (1997). The idea is to provide the designers of multiprocess programs with a language construct (namely, the notion of an atomic procedure called a *transaction*) that discharges them from the management of synchronization issues. More precisely, a programmer has to concentrate her efforts only on defining which parts of processes have to be executed atomically and not on the way atomicity is realized, this last issue being automatically handled by the underlying STM system.

This chapter, which is mainly on basic principles of algorithms implementing STM systems, assumes that the asynchronous processes are reliable (i.e., they never crash).

Keywords Abort · Atomic execution unit · Commit · Deferred updates · Incremental read (snapshot) · Lock · Multi-version · Opacity · Read invisibility · Read/modify/write · Software transactional memory · Speculative execution · Transaction (read-only, write-only, update) · Virtual world consistency

10.1 What Are Software Transactional Memories

10.1.1 Transactions = High-Level Synchronization

As we have seen in the previous chapters, the implementation of concurrent objects can be lock-based or mutex-free. The design of such implementations is not always an easy task. Hence, the following idea:

- Provide programmers with efficient implementations of base atomic objects such as stacks, queues, counters, snapshots, etc., these implementations being kept in a library (and the way they are realized being hidden to programmers).

- Provide programmers with a *scalable* and *easy-to-use* language construct which allows them to define which parts of code involving accesses to base objects have to be executed atomically.

The central idea is that programmers have to concentrate only on which parts of their multiprocess programs have to be executed atomically. The aim of such a language construct is consequently to discharge the programmer from the direct-management of the synchronization entailed by concurrent accesses to atomic objects. Moreover, *scalable* means that the programmer is not restricted in the number of base atomic objects that have to appear as being simultaneously accessed in an atomic way.

Locks cannot be composed It is important to observe that locks are not scalable and cannot be composed. They consequently cannot constitute the answer to the previous issue.

As an example, let us consider a multiprocess program in which processes access two atomic queues $Q1$ and $Q2$ with the operations enqueue() and dequeue(). Moreover, let us consider that a process wants to move atomically the first item of $Q1$ into $Q2$. To that end the following procedure move_item($Q1$, $Q2$) could be defined:

```
procedure move_item($Q1$, $Q2$) is
        $x \leftarrow Q1$.dequeue(); $Q2$.enqueue($x$)
end procedure.
```

Unfortunately, this does not work: the move is not atomic as, between the invocation of $Q1$.dequeue() and the invocation of $Q2$.enqueue(x), an arbitrary number of accesses to $Q1$ and $Q2$ can be issued by other transactions.

A first answer to this problem could be to define a polyadic procedure move_item ($Q1$, $Q2$) as a base atomic operation on queues (the procedure is polyadic because it is on two objects—here queues—at the same time).This ad hoc solution is not satisfactory for the following reasons:

- It can make the implementation of a queue more difficult (polyadic atomic operations are more difficult to implement than unary operations).
- It is not scalable. If a procedure has to be executed atomically on a queue and a stack, where the corresponding operation is defined? In the definition of the stack? In the definition of the queue? Moreover, how this can be done if the procedure, that has to appear as being executed atomically, involves more base objects than only two, or if this number is not statically defined?

Actually, the part of code that has to be executed atomically is not part of the definition of the objects but is application-dependent.

A language construct What is needed is a language construct saying that the code corresponding to some procedure has to appear as having been executed atomically. This is exactly what the concept of a transactional memory offers to programmers.

Basically, the previous procedure move_item($Q1$, $Q2$) has to be defined as follows:

```
transaction move_item($Q1$, $Q2$) is
        $x \leftarrow Q1$.dequeue(); $Q2$.enqueue($x$)
end transaction,
```

where the keyword **transaction** means that the associated code has to appear as being executed as an atomic operation. It is the job of the underlying *software transactional system* (STM) to ensure this atomicity property.

STM transaction versus database transaction While transactions defined in an STM system and transactions encountered in a database share the same name, they are different computer science objects. A main difference lies in the following observation: the code of an STM transaction can be any code (accessing base atomic objects), while the code of a database transaction is usually restricted to SQL-like queries. STM transactions are actually *atomic procedures*.

The high-level view Moving beyond assembly languages, the design of high-level programming languages was mainly motivated by hiding "implementation details" to allow the programmer to concentrate on the solution to his problem and not on the technicalities of specific machines or on low-level machineries. As an example, garbage collection is now implicit in a lot of programming languages and the programmer does not have to worry about it.

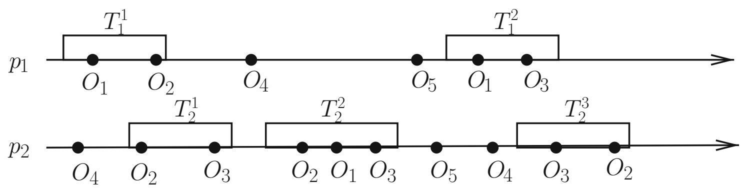
Transactional memories constitute a similar effort as far as synchronization is concerned. The programmer has to state what has to be atomically executed and has not to focus on the way this synchronization has to be realized. This is the job of the underlying STM system.

10.1.2 At the Programming Level

The program written by the user is made up of n sequential processes denoted $p_1, \ldots, p_n$. Each process is a sequence of transactions in which two consecutive transactions can be separated by non-transactional code. Both transactions and non-transactional code can access atomic objects (called application objects).

Transactions A transaction is an atomic unit of computation (atomic procedure) that can access atomic objects called t-objects. "Atomic" means that (from the programmer's point of view) a transaction appears as being executed instantaneously at a single point of the time line (between its start event and its end event) and no two transactions are executed at the same point of the time line. It is assumed that, when executed alone, a transaction always terminates.

Conceptually a transaction can be seen as a programmer-defined *read/modify/write* operation. It first obtains values from some atomic objects, then does local computation, and finally assigns new values to atomic objects.



T_1^1
T_1^2
p_1
O_1 O_2 O_4 O_5 O_1 O_3
T_2^1
T_2^2
T_2^3
p_2
O_4 O_2 O_3 O_2 O_1 O_3 O_5 O_4 O_3 O_2

Fig. 10.1 An execution of a transaction-based two-process program

Non-transactional code Non-transactional code is made up of statements that the user does not require to appear as being executed as a single atomic computation unit. This code usually contains input/output statements (if any). Non-transactional code can also access atomic objects. These objects are called nt-objects.

It is assumed that an atomic object is either an nt-object or a t-object (not both).

Example Let us consider Fig. 10.1, which describes an execution of a two-process program (p_1 and p_2) accessing five atomic objects: O_1, O_2, and O_3, which are t-objects (accessed inside transactions) and O_3 and O_5, which are nt-objects (accessed outside transactions).

Process p_1 first issues transaction T_1^1 (which internally accesses O_1 and O_2) (the transaction is depicted as a rectangle and its accesses to objects are depicted with black dots), then accesses directly O_4 and O_5, and finally issues transaction T_1^2 (which accesses O_1 and O_5). Process p_2 accesses first O_4, then issues T_2^1 (which accesses O_2 and O_3) followed by T_2^2 (which accesses O_2, O_1 and O_3), then accesses O_5 and O_4, and finally issues transaction T_2^3 (which accesses O_3 and O_4).

As indicated, each transaction has to appear as if it was executed at a single point of the time line. An example of correct behavior as seen by an external observer is described in Fig. 10.2. This observer sees first the access of p_2 to object O_4, then the execution of transaction T_2^1 by p_2, then the execution of transaction T_1^1 by p_1, etc.

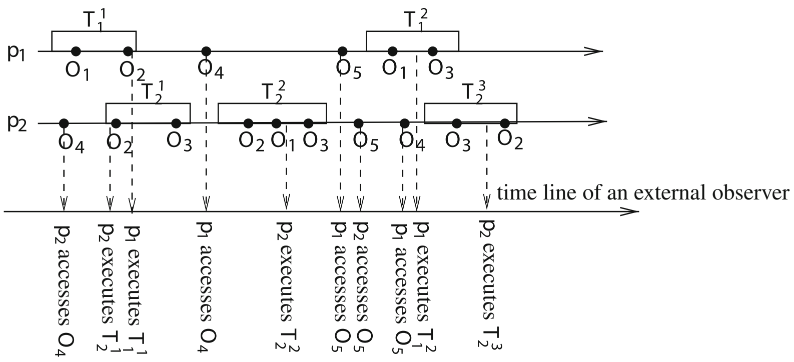


Fig. 10.2 Execution of a transaction-based program: view of an external observer

10.2 STM System

To simplify the presentation, the rest of this chapter considers that the processes do not contain non-transactional code and that the atomic objects shared by transactions are MWMR registers.

10.2.1 Speculative Executions, Commit and Abort of a Transaction

Speculative execution Like a scheduler, an STM system is an online algorithm that does not know the future. Its aim is to ensure that the transactions issued by the processes of a multiprocess program are executed atomically. One way to ensure this property would consist in allowing one transaction at a time to execute, but this would be very inefficient. Hence, to be efficient, STM systems have to permit several transactions to execute simultaneously. Such transaction executions are called *speculative* (or *optimistic*) executions.

Commit versus abort Speculative executions of transactions have a price. This is because it may happen that it is impossible to totally order in a correct way two (or more) transactions that have been executed speculatively. This occurs if these transactions conflict: both access the same base object and one of them modifies its value.

As a very simple example let us consider a single base object X which is an atomic MWMR register initialized to 0, and the transactions T_1 and T_2 speculatively executed in parallel by p_1 and p_2, respectively, where

transaction T_1 **is** $x \leftarrow X + 1; X \leftarrow x$ **end transaction**.
transaction T_2 **is** $x \leftarrow X + 2; X \leftarrow x$ **end transaction**.

It is easy to see that, if T_1 and T_2 are executed in parallel, the final value of X can be 1, 2 or 3. But, as a transaction is an atomic execution unit, the only correct value for X after both transactions have been executed is the one produced by T_1 followed by T_2 or T_2 followed by T_1, i.e., the value 3.

This means that, when a speculative execution of a transaction is about to terminate, it is required to check if it can be linearized at some point of the time line. If it can, it is committed, otherwise it is aborted. Hence, a speculative execution of a transaction has to return a control value *commit* or *abort* that defines its fate. When a transaction execution is aborted, the STM system may decide to re-execute it or notify the corresponding process, which decides to re-issue it or not.

It is important to notice that there is always a price to pay when processes access shared objects. This price is either **wait** statements or re-execution. In both cases, it means that time duration without progress cannot be prevented. This is an inescapable price that has to be paid when there is synchronization.

10.2.2 An STM Consistency Condition: Opacity

Preventing reading from inconsistent global states As already indicated, the code of an STM transaction is not restricted to predefined patterns; it can be any piece of code. It follows that a transaction has to always operate on a consistent state of the objects it accesses. To motivate this claim, let us consider a transaction T that contains the statement $x \leftarrow A/(B-C)$, where A, B, and C are three atomic MWMR registers and the predicate $B \neq C$ is true in all the consistent global states of the multiprocess program. If the values of B and C come from different global states, it is possible that the transaction T obtains values such as $B = C$ (which is inconsistent). If this occurs, the transaction T raises a "division by 0" exception that has to be handled by the process that issued the transaction T. Other bad behaviors can be produced when a transaction reads values from an inconsistent state and these mutually inconsistent values generate infinite loops inside a transaction.

Such bad behaviors have to be prevented by an STM system: whatever its fate (commit or abort) a transaction has to always obtain mutually consistent values.This is captured by the *opacity* consistency condition, which states that no transaction reads values from an inconsistent global state.

Opacity Let us associate with each aborted transaction T the read prefix that contains all its read operations of base MWMR registers until T aborts (if the abort is entailed by a read, this read is not included in the read prefix).

An execution of a multiprocess program satisfies the *opacity* consistency condition if all the committed transactions and the read prefix of each aborted transaction appear as if they have been executed one after the other, this sequential order being in agreement with their real-time occurrence order.

10.2.3 An STM Interface

An STM system interface for transactions that access MWMR atomic registers provides each transaction with four operations, denoted $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$, where T is a transaction, and X an atomic t-object (a MWMR register shared by the transactions).

- $\mathsf{begin}_T()$ is invoked by T when it starts. It initializes local control variables.
- $X.\mathsf{read}_T()$ is invoked by the transaction T to read the base object X. That operation returns a value of X or the control value $abort$. If $abort$ is returned, the invoking transaction is aborted (in that case, the corresponding read does not belong to the read prefix associated with T).
- $X.\mathsf{write}_T(v)$ is invoked by the transaction T to update X to the new value v. That operation returns the control value *ok* or the control value $abort$. Like in the operation $X.\mathsf{read}_T()$, if $abort$ is returned, the invoking transaction is aborted.

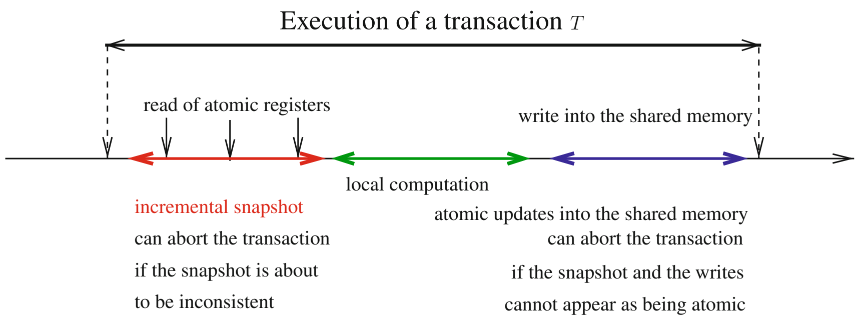

Execution of a transaction T
read of atomic registers
write into the shared memory
local computation
incremental snapshot
can abort the transaction
if the snapshot is about
to be inconsistent
atomic updates into the shared memory
can abort the transaction
if the snapshot and the writes
cannot appear as being atomic

Fig. 10.3 Structure of the execution of a transaction

- If a transaction reaches its last statement, it invokes the STM interface operation $\text{try\_to\_commit}_T()$. That operation decides the fate of T by returning $commit$ or $abort$. (Let us notice that a transaction T that invokes $\text{try\_to\_commit}_T()$ has not been aborted during an invocation of $X.\text{read}_T()$ or $X.\text{write}_T()$.)

10.2.4 Incremental Reads and Deferred Updates

In the transaction system model we considered here, each transaction T uses a local working space. When T invokes $X.\text{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later invocations of $X.\text{read}_T()$ (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory may have been changed in between by other transactions. Such incremental reads are also called *incremental snapshots*.

When T invokes $X.\text{write}_T(v)$, it writes v into its working space (and does not access the shared memory). Finally, if T is not aborted when it executes $\text{try\_to\_commit}_T()$, it copies (if any) the values of application registers it has written from its local working space into the shared memory. (A similar deferred update model is used in some database transaction systems.)

The corresponding structure of the execution of a transaction is represented in Fig. 10.3.

10.2.5 Read-Only Versus Update Transactions

When the atomic application objects are atomic MWMR registers (which is our case), a transaction that issues only read operations on application objects is called a *read-only* transaction. Otherwise, it is called an *update* transaction. (Hence an update transaction is either a read/write transaction or a write-only transaction.)

10.2.6 Read Invisibility

Read invisibility is an implementation property of an STM system that concerns the operation $X.\mathsf{read}_T()$.

An algorithm implementing $X.\mathsf{read}_T()$ may or may not be able to write control information into the shared memory. If it writes such information (e.g., the identity of the corresponding transaction), the implementation of $X.\mathsf{read}_T()$ is said to be *visible*. Otherwise, it is *invisible*.

10.3 A Logical Clock-Based STM System: TL2

This section presents a simplified version of a lock-based STM system called TL2 (Transactional Locking 2) due to D. Dice, O. Shalev, and N. Shavit (2006). This STM system satisfies the opacity consistency condition. It is particularly efficient when there are few conflicts between concurrent transactions.

10.3.1 Underlying System and Control Variables of the STM System

Underlying system The underlying system provides the processes with atomic MWMR registers and an atomic fetch&add register. In addition to a read operation, such a register A allows the processes to invoke the operation $\mathsf{fetch\&add}()$ which atomically adds 1 to A and returns the new value of A to the invoking process.

Global control variable An atomic fetch&add register denoted $CLOCK$, initialized to 0, is used as a logical clock to measure the progress of the system, counted as the number of transactions that have been committed so far.

As we will see, the abort/commit decision for a transaction T will involve this clock and the dates associated with the application registers accessed by transaction T.

Internal representation of application registers and associated control variables As already indicated, the application registers are the MWMR atomic registers defined at the application level and accessed by the transactions.

At the system level, an implementation MWMR register XX located in the shared memory is associated with each application MWMR register X. Such an implementation register XX has two fields: $XX.value$, which contains the value of the application register X, and $XX.date$, which contains the date of its last update. A lock is also associated with each implementation register XX.

As far as notations are concerned, $lc(XX)$ denotes a copy of XX in the local working space of a process.

Local control variables associated with a transaction Each process p_i maintains two local variables denoted $lrs(T)$ and $lws(T)$, where T is the transaction that it is currently executing. $lrs(T)$ (local read set of T) contains the names of all the application registers X that T has read up to now. Similarly, $lws(T)$ (local write set of T) contains the names of all the application registers that T has written up to now in its local working space.

Moreover, when a transaction T is started by a process p_i (or re-started after an abort), its birth date (which is defined from the current value of $CLOCK$) is saved by p_i in its local variable $birthdate(T)$.

10.3.2 Underlying Principle: Consistency with Respect to Transaction Birth Date

The idea is to commit a transaction T if it could appear to an external observer as if it was entirely executed at the (logical) time $birthdate(T)$. When this is not possible, the transaction is aborted. To that end, the management of the clock is such that, when the transaction T starts its execution, the last update issued by committed transactions of every application register X is such that $XX.date < birthdate(T)$ (see Fig. 10.4).

Validation test associated with incremental read Let us consider a transaction T that wants to read an application register X. There are two cases according to the value of $XX.date$:

- If $XX.date < birthdate(T)$, the value read by transaction T was present in shared memory when T was created. Hence, the read is consistent and T can continue its execution. The incremental read of X is successful. As we can see from Fig. 10.4, if T reads only X, Y, and Z, these reads return values which are mutually consistent in the sense that, despite the fact that they were read at different times, they were simultaneously present when T was created.
- If $XX.date \geq birthdate(T)$, the value of X read by T was written after T was created. As a transaction cannot read from the future, it is not possible to consider that T was atomically executed at the date $birthdate(T)$. Hence, when this occurs, the incremental read of X is no longer consistent with the previous reads issued by the transaction T, which is consequently aborted.

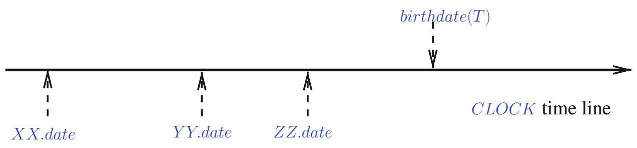


Fig. 10.4 Read from a consistent global state

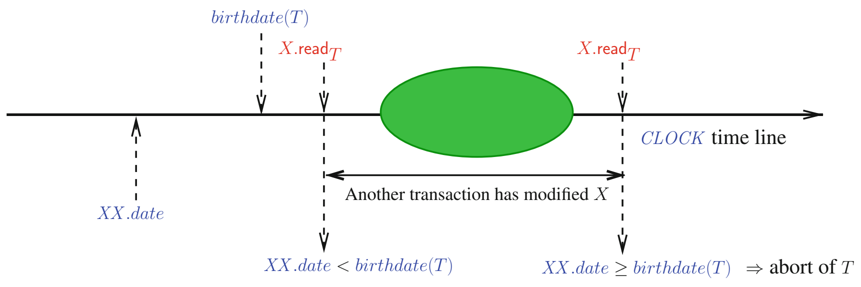


Fig. 10.5 Validation test for a transaction T

Final validation test Let us consider a transaction T that has not been aborted during its incremental read phase, has done local computation, and wants to write new values into some application register Z.

For T to appear as having been executed atomically at the logical date $birthdate(T)$, the date that T has to associate with the values it wants to write has to be equal to $birthdate(T)$ (this date is obtained after increasing the clock by 1).

But it is possible that another transaction has modified an application register X after it was read by T. If this is the case, the date associated with the new value of X will be greater than $birthdate(T)$. Hence, the reads of application registers issued by T would appear as being done at the date $birthdate(T)$ while its write of new values will appear as being done at a date $> birthdate(T)$. It follows that T cannot appear as having been executed atomically. When this occurs, T is aborted (Fig. 10.5).

From an operational point of view, a transaction T that executes $\mathsf{try\_to\_commit}_T()$ is required to read again from the shared memory the dates of the application registers X it has previously read during the incremental read phase. If there is an application register X such that $XX.date \geq birthdate(T)$, the transaction T is aborted.

Remark It is important to see that the consistency predicates which are used to satisfy opacity are based on comparison on dates defined from the logical time produced from the atomic register $CLOCK$. This logical time is sufficient to obtain opaque implementations. It is not necessary.

10.3.3 The Implementation of an Update Transaction

The algorithms implementing the operations of the STM interface are described in Fig. 10.6. Let p_i be the process that has issued the transaction T.

The operation $\mathsf{begin}_T()$ When a process p_i issues a transaction T, it first invokes $\mathsf{begin}_T()$ to assign a birth date to T.

```
operation begin_T() is
(1)  $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$; $birthdate(T) \leftarrow CLOCK + 1$; return()
end operation.

operation $X$.read_T() is
(2)  if (there is a local copy $lc(XX)$ of $XX$)
(3)     then return($lc(XX).value$)
(4)     else $lc(XX) \leftarrow$ copy of $XX$ read from the shared memory;
(5)          if ($lc(XX).date < birthdate(T)$)
(6)               then $lrs_T \leftarrow lrs_T \cup \{X\}$; return($lc(XX).value$)
(7)               else return($abort$)
(8)          end if
(9)  end if
end operation.

operation $X$.write_T($v$) is
(10) if (there is no local copy of $XX$) then allocate local space $lc(XX)$ for a copy end if;
(11) $lc(XX).value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$;
(12) return($ok$)
end operation.

operation try_to_commit_T() is
(13)   lock all the objects in ($lrs_T \cup lws_T$);
(14)   for each $X \in lrs_T$ do % the date of $XX$ is read from the shared memory %
(15)        if $XX.date \geq birthdate(T)$ then release all the locks; return($abort$) end if
(16)   end for;
(17)   $write\_date \leftarrow CLOCK$.fetch&add();
(18)   for each $X \in lws_T$ do $XX \leftarrow (lc(XX).value, write\_date)$ end for;
(19)   release all the locks; return ($commit$)
end operation.
```

Fig. 10.6 TL2 algorithms for an update transaction

The operation X.read$_T$() This operation is invoked by p_i each time the transaction T issues a read of the application register X. If there is a local copy $lc(XX)$ of X, the value of X stored in $lc(XX).value$ is returned (lines 2–3). Otherwise, a local copy is created and initialized to the value of XX read from the shared memory (line 4). Finally, the validation test associated with the incremental read of X is done (line 5). If it is successful, the current value of X is returned and X is added to lrs_T (line 6). Otherwise, T is aborted (line 7).

It is important to notice that this implementation of X.read$_T$() satisfies the *read invisibility* property: no information is written into the shared memory by an invocation of X.read$_T$().

The operation X.write$_T$(v) This operation is invoked each time the transaction T, issued by p_i, writes a new value into the application register X. Actually, this algorithm does not write the new value into the shared memory but only in the local copy $lc(XX).value$ (line 11). Such a local copy is previously created if there is no local copy (line 10). Moreover, the set lws_T is updated accordingly (line 11). It is easy to see that no invocation of X.write$_T(v)$ entails the abortion of T.

The operation try_to_commit$_T()$ This operation is called by p_i when T has reached its last statement without having been previously aborted. This means that all the incremental reads issued by T are consistent with the birth date of T. As we have seen, it remains to see if the writes of the application registers issued by T can appear as having been atomically done at the date $birthdate_T$ (i.e., together with the incremental reads issued by T).

When it executes try_to_commit$_T()$, p_i first locks both the application registers that have been read by T and the the ones that will be written by T (line 13). (In order to prevent deadlocks, it is assumed that these lockings are done sequentially, according to a predefined total order.)

Then, p_i executes the final validation test (as explained above). To that end, for each $X \in lrs_T$, it reads the current value of the implementation register XX and aborts the transaction if there is some X such that $XX.date \geq birthdate(T)$ (line 15). If T is aborted, all the locks are released.

If the final validation test succeeds, the transaction T can be committed. But before releasing the locks and committing T (line 19), p_i has first to compute the new date that has to be associated with all the writes issued by T (line 17). It can then issue the corresponding writes into the shared memory (line 18).

On the use of locks It is important to notice that locks are used only in the operation try_to_commit$_T()$. Hence, an implementation register can be read by a transaction while it is locked by another transaction. The use of the fetch&add register *CLOCK* ensures that no two committed transactions associate the same date with their writes. The locking of all the registers accessed by a transaction T (whose names are saved in $lrs_T \cup lws_T$) ensure that no date of a register XX can be modified while p_i is checking the final validation text (lines 14–18), thereby ensuring that, from an external observer's point of view, all the writes into the shared memory of the registers in lws_T (line 18) appear as having been executed (a) at the date $birthdate_T$ and (b) simultaneously with the read of the registers in lrs_T.

Remark This presentation of TL2 does not take into account all its features. As an example, if at line 13 a lock cannot be immediately obtained, TL2 aborts the corresponding transaction. This can allow for a more efficient implementation.

10.3.4 The Implementation of a Read-Only Transaction

If a transaction T does not write application registers, its STM interface can be simplified as shown in Fig. 10.7. Without loss of generality, it is assumed that a read-only transaction reads an application register at most once.

A local copy no longer plays the role of a local cache memory. When T invokes X.read$_T()$, XX is read from the shared memory (line 2) and the associated incremental

```
operation begin_T() is
 (1) $birthdate_T \leftarrow CLOCK + 1$; return()
end operation.

operation $X$.read_T() is
 (2) $lc(XX) \leftarrow$ copy of $XX$ read from the shared memory;
 (3) if $lc(XX).date \geq birthdate_T$ then return($abort$) else return($lc(XX).value$) end if
end operation.

operation try_to_commit_T() is
 (4) return($commit$)
end operation.
```

Fig. 10.7 TL2 algorithms for a read-only transaction

read validation test is executed (line 3). This modified version of X.read$_T$() satisfies the read invisibility property.

As T does not write application registers, the lines 17–18 of try_to_commit$_T$() in Fig. 10.6 become useless. Moreover, the final validation test of line 15 of Fig. 10.6 is now done each time p_i executes X.read$_T$(). It follows that locks are no longer needed in the operation try_to_commit$_T$(), which boils down to a simple invocation of the statement return($commit$) (line 4).

Interestingly, these algorithms satisfy a strong read invisibility property. Not only does X.read$_T$() guarantee that there is no visible read, but even try_to_commit$_T$() does not reveal which registers have been read by a read-only transaction.

10.4 A Version-Based STM System: JVSTM

This section presents an STM system based on multi-versionning that satisfies the opacity property. Multi-version means that an application MWMR atomic register is implemented by a list of versions containing its successive values. This implementation, called JVSTM (Java STM), is due to J. Cachopo and A. Rito-Silva (2006). As for TL2, the version which is presented here is a version which has been simplified for pedagogical purposes.

Interestingly, JVSTM provides a garbage collector mechanism that discards the versions older than the oldest transaction which are still in the system. This point is not addressed here.

10.4.1 Underlying and Control Variables of the STM System

Underlying system The underlying system is made up of implementation MWMR atomic registers and a starvation-free lock object denoted $LOCK$. This unique lock will be used by the operation try_to_commit$_T$() to ensure that no two transactions are committed at the same time.

Global control variable In order to be able to exploit the different versions of the application registers that have been created by the committed transactions, this STM system uses a logical clock (denoted $CLOCK$) to timestamp the transactions and allow each of them to read the appropriate version of the corresponding register.

$CLOCK$ is an implementation MWMR atomic register initialized to 0. It is increased by a transaction T when that transaction is about to commit. Its updated value is then used to associate a date with the new cell associated with each application register that is written by T.

Internal representation of an application register X At the implementation level, an application register X is represented by a pointer $PT[X]$ on a list of cells. Each cell CX is made up of three fields:

- $CX.value$ is a data field containing a value of X written by a committed transaction,
- $CX.date$ is a control field containing the date at which that value was written, and
- $CX.prev$ is a pointer to the previous cell in the list.

The list of versions associated with an application register X is depicted in Fig. 10.8, where the application register X is initialized to the value x_0 (and the corresponding date is 0).

Let us remember that the following notations are used: if Y contains a pointer, $(\downarrow Y)$ denotes the cell pointed to by Y, and If C is a cell, $(\uparrow C)$ denotes a pointer to C.

Local control variables associated with a transaction The local variables $birthdate_T$, lrs_T and lws_T have the same meaning as in the previous section devoted to the STM system TL2.

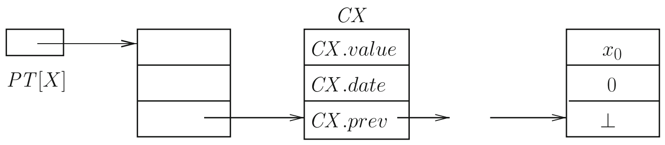


Fig. 10.8 The list of versions associated with an application register X

10.4.2 The Implementation of an Update Transaction

A description of the X.read$_T$(), X.write$_T(v)$, and try_to_commit$_T$() operations for an update transaction is given in Fig. 10.9. As in the previous section, p_i is the process that has issued the transaction T.

The operation X.begin$_T$() Similarly to TL2, this operation consists in computing the birth date of the transaction T.

```
operation $X$.begin$_T$() is
(1)   $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$; $birthdate_T \leftarrow CLOCK$; return()
end operation.

operation $X$.read$_T$() is
(2)   if ($\nexists$ local copy $lc(CX)$ of a cell $CX$ associated with $X$)
(3)     then $lrs_T \leftarrow lrs_T \cup \{X\}$;
(4)          $ptx \leftarrow PT[X]$;
(5)          while $((\downarrow ptx).date > birthdate_T)$ do $ptx \leftarrow (\downarrow ptx).prev$ end while;
(6)          allocate local space $lc(CX)$; $lc(CX) \leftarrow (\downarrow ptx)$;
(7)   end if;
(8)   return($lc(CX).value$)
end operation.

operation $X$.write$_T(v)$ is
(9)   if ($\nexists$ local copy $lc(CX)$ of a cell associated with $X$) then allocate space $lc(CX)$ end if;
(10)  $lc(CX).value \leftarrow v$;
(11)  $lws_T \leftarrow lws_T \cup \{X\}$;
(12)  return($ok$)
end operation.

operation try_to_commit$_T$() is
(13)  $LOCK$.acquire_lock();
(14)  if $(lws_T \neq \emptyset)$ then
(15)    for each $X \in lrs_T$ do
(16)         if $((\downarrow PT[X]).date > birthdate_T)$ then release lock; return $(abort)$ end if
(17)    end for
(18)  end if;
(19)  $commit\_date \leftarrow CLOCK + 1$;
(20)  for each $X \in lws_T$ do
(21)    $lc(CX).prev \leftarrow PT[X]$; $lc(CX).date \leftarrow commit\_date$;
(22)    allocate a new cell $CCX$ in the shared memory; $CCX \leftarrow lc(CX)$;
(23)    $PT[X] \leftarrow (\uparrow CCX)$
(24)  end for;
(25)  $CLOCK \leftarrow commit\_date$;
(26)  $LOCK$.release_lock();
(27)  return $(commit)$
end operation.
```

Fig. 10.9 JVSTM algorithm for an update transaction

The operation $X.\mathsf{read}_T()$ The algorithm implementing the operation $X.\mathsf{read}_T()$ is as follows. If there is already a local copy $lc(CX)$ of a cell associated with the application register X, p_i returns the value $lc(CX).value$.

Otherwise, considering the list associated with X (line 4), p_i searches for the most recent cell CX containing a value whose writing date is compatible with $birthdate_T$, i.e., such that $CX.date \leq birthdate_T$ (line 5). It then creates a local copy $lc(CX)$ of that shared memory cell (line 6) and returns its data value (line 8).

As in TL2, it is easy to see that this implementation of $X.\mathsf{read}_T()$ satisfies the read invisibility property.

The operation $X.\mathsf{write}_T(v)$ The operation $X.\mathsf{write}_T(v)$ is similar to the one of TL2. If there is no local copy $lc(CX)$ of a cell associated with X, one is first created (line 9). Then, the value v is assigned to $lc(CX).value$ and the local write set lws_T is updated accordingly (lines 10–11).

The operation $\mathsf{try\_to\_commit}_T()$ When a transaction T invokes $\mathsf{try\_to\_commit}_T()$, the corresponding process p_i first acquires the global lock (line 13). Then, if $lws_T \neq \emptyset$, p_i checks if the current value of each object X read by T is the most recent one (lines 15–17). This is to check that the read and write operations issued by T can appear as being executed atomically at the date $CLOCK + 1$. This is done by comparing the current date $(\downarrow PT[X]).date$ of each object X that was read with the birth date of T (line 16). If one of these dates is greater than the birth date of T, it will not be possible for both the reads issued by T (which appear as being simultaneously executed at the date $birth\_date$) and its writes to appear as being atomically executed at the date $CLOCK + 1$. If this is the case, T is aborted (line 16). Otherwise, it it committed.

Before committing T, the date of the write of the new values of the registers X locally written by T (operation $X.\mathsf{write}_T()$) is first computed (line 19), and then the new values are written in new cells allocated in the shared memory (lines 21–22). Finally, for each X that has been written, the global pointer $PT[X]$ is appropriately updated (line 23).

Remark 1 Let us observe that a write-only transaction T never aborts. This is because, due to the lock that protects both the register $CLOCK$ and the objects in lws_T, the write operations of T (a) appear as being logically executed at the same date ($commit\_date$) and (b) do not have to appear as being logically executed with read operations.

Remark 2 This presentation of the $\mathsf{try\_to\_commit}_T()$ operation of JVSTM does not take into account all of its aspects. Among those, there is a mechanism that keeps track of the birth dates of the transactions that are not yet committed. When a transaction is committed, it discards all the versions of the registers that it wrote which are no longer accessible (this is because these versions are too old to be read by any live transaction). The JAVA garbage collector then frees the corresponding memory locations.

```
operation $X.\mathsf{init}_T()$ is
(1) $birthdate_T \leftarrow CLOCK$: return()
end operation.

operation $X.\mathsf{read}_T()$ is
(2) $ptx \leftarrow PT[X]$;
(3) while $\big((\downarrow ptx).date > birthdate_T\big)$ do $ptx \leftarrow (\downarrow ptx).next$ end while;
(4) return$\big((\downarrow ptx).value\big)$
end operation.

operation try_to_commit$_T()$ is
(5) return $(commit)$
end operation.
```

Fig. 10.10 JVSTM algorithm for a read-only transaction

10.4.3 The Implementation of a Read-Only Transaction

If a transaction T does not modify the application registers, the algorithms implementing the interface operations of the STM system can be simplified as shown in Fig. 10.10. Without loss of generality, it is assumed that a read-only transaction reads an application register at most once.

Each time a read-only transaction T reads an application register X, it obtains (from the shared memory) the most recent value which is at least as old as the transaction's birth date (lines 2–4). A read-only transaction is not required to manage a local read set lrs_T.

As in TL2, the operation try_to_commit$_T()$ always returns $commit$. This is because a read-only transaction can never abort. This is due to the presence of multiple versions which always allow a read-only transaction T to obtain mutually consistent versions of the application registers it reads.

10.5 A Vector Clock-Based STM System

10.5.1 The Virtual World Consistency Condition

Virtual world consistency While keeping its spirit, *virtual world consistency* is a weaker consistency condition than opacity. It states that (a) no transaction (committed or aborted) reads values from an inconsistent global state, (b) the committed transactions are linearizable (they can be totally ordered from an external observer point of view), and (c) each aborted transaction (reduced to a read prefix as defined in Sect. 10.2.2 where opacity was introduced) reads values that are consistent with respect to its causal past only.

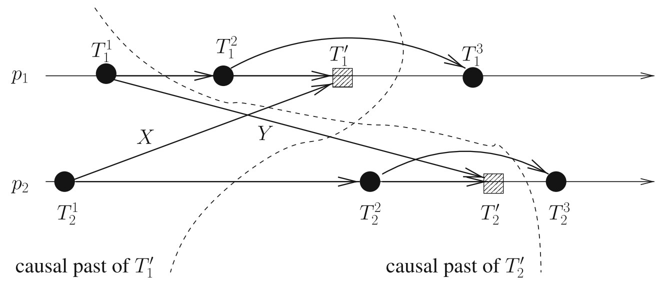


Fig. 10.11 Causal pasts of two aborted transactions

The causal past of an aborted transaction T is defined as follows. Let T' be a committed transaction that has written a value that was read by T, and let T'' be any committed transaction that appears before T' in the total order including all committed transactions. The causal past of T includes all these transactions T' and T''.

As two aborted transactions can have different causal pasts, each can read from a global state that is consistent from its causal past point of view, but the global states from which two aborted transactions have read can be mutually inconsistent, as these two aborted transactions do not necessarily have the same causal past (hence the name *virtual world* consistency).

An example To better understand the intuition that underlies virtual world consistency, let us consider the transaction execution depicted in Fig. 10.11. There are two processes: p_1 has sequentially issued T_1^1, T_1^2, T_1', and T_1^3, while p_2 has issued T_2^1, T_2^2, T_2', and T_2^3. The transactions associated with a black dot have committed, while the ones with a grey square have aborted. From a data dependency point of view, each transaction T issued by a process p_i depends on its previously committed transactions (internal dependency) and on the committed transactions T_x issued by other processes from which T has read values (external dependency) and, recursively, on the transactions T_y on which each previous T_x depends.The label X on the edge from T_2^1 to T_1' means that T_1' has read from X a value written by T_2^1. In contrast, since an aborted transaction does not write, there is no dependency edges originating from it.

The causal past of the aborted transactions T_1' and T_2' are indicated on the figure (including the committed transactions which appear on the left side of the corresponding dotted line). The values read by T_1' are consistent with respect to its causal past dependencies, and similarly for T_2'. It is nevertheless possible that it is impossible to totally order T_1' and T_2' with all committed transactions.

It follows that, while opacity implies virtual world consistency, the converse is not true. The noteworthy feature of the virtual world consistency condition is that it can allow more transactions to commit than opacity while keeping its spirit.

10.5.2 An STM System for Virtual World Consistency

This section presents an STM system that satisfies the virtual world consistency condition. This system, due to D. Imbs and M. Raynal (2009), is based on vector clocks and guarantees the read invisibility property.

While n denotes the number of processes, m is used to denote the number of application MWMR atomic registers.

Internal representation of the application MWMR atomic registers Each application MWMR atomic register X is represented by an implementation MWMR atomic register XX made up of two fields:

- $XX.value$ contains the current value of X.
- $XX.depend[1..m]$ is a vector clock which tracks value dependencies. More precisely,
  - $XX.depend[X]$ contains the sequence number associated with the current value of X, and
  - $XX.depend[Y]$ contains the sequence number associated with the value of Y on which the current value of X depends.

  Such a vector is called a *vector clock* because, sequence numbers being considered as logical dates associated with updates, $depend[1..m]$ captures the causal dependences among these updates.

Moreover, a starvation-free lock is associated with each application register X.

Local control variables associated with processes and transactions A process issues transactions sequentially. So, when a process p_i issues a new transaction, that transaction has to work with object values that are not older than the ones used by the previous transactions issued by p_i. To that end, p_i manages a local vector $p\_depend_i[1..m]$ such that $p\_depend_i[X]$ contains the sequence number of the last value of X that (directly or indirectly) is known by p_i.

In addition to the previous array, a process p_i manages the following local variables whose scope is the one of the transaction T it is currently executing:

- $t\_depend_T[1..m]$ is a copy of $p\_depend_i[1..m]$ which is used instead of it during the speculative execution of T (this is because $p\_depend_i[1..m]$ must not be modified if T aborts).
- lrs_T and lws_T are the read set and write set used when p_i executes T.
- Finally, for each application register X accessed by T, p_i manages a local copy denoted $lc(XX)$ of the implementation register XX.

10.5.3 The Algorithms Implementing the STM Operations

This section presents the algorithms implementing the four STM operations $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$ (Fig. 10.12) and some of their properties. When the control value *abort* is returned, it carries a tag (1 or 2) which indicates the cause of the abortion of the corresponding transaction. This tag is used only for pedagogical purposes.

The operation $\mathsf{begin}_T()$ This operation is a simple initialization of the local control variables associated with the current transaction T. Let us notice that t_depend_T is initialized to p_depend_i to take into account the causal dependencies on the values accessed by the committed transactions previously issued by p_i. This is due to the fact that a process p_i issues transactions one after the other and the next one inherits the causal dependencies created by the previous ones.

The validation test for an incremental read and the operation $X.\mathsf{read}_T()$ This operation returns either a value of X or the control value *abort* (in which case T is aborted). If (due to a previous read of X) there is a local copy, its value is returned (lines 2 and 10).

If the call $X.\mathsf{read}_T()$ is its first read of X, p_i first builds a copy $lc(XX)$ from the shared memory (line 3), and updates accordingly its local control variables lrs_T and $t\_depend_T[X]$ (line 4). Hence, $t\_depend_T[X]$ contains the sequence number associated with the last value of X which was saved in $lc(XX).value$.

As the reads are incremental (p_i does not read in one atomic action all the application registers it wants to read), p_i has to check that (a) the value it has just read from the shared memory and stored in $lc(XX).value$ and (b) the values of the application registers Y it has previously read can belong to a consistent global state.

The corresponding incremental read test is done as follows. Let Y be an object that was previously read by T (hence $Y \in lrs_T$). Let us observe that the sequence number of the value of Y read by T is kept in $t\_depend_T[Y]$. If the value of X just read by T depends on a more recent value of Y, the values of X and Y are mutually inconsistent. This is exactly what is captured by the predicate

$$\exists\, Y \in lrs_T : \big(t\_depend_T[Y] < lc_i(X).depend[Y]\big)$$

used at line 5. If this predicate is true, p_i aborts T. Otherwise, p_i first updates $t\_depend_T[1..m]$ for all the application registers $Y \notin lrs_T$ (lines 6–8) to take into account the new dependencies (if any) created by the reading of X. Finally, the value obtained from $lc(XX)$ is returned (line 10).

It is easy to see that an invocation of $X.\mathsf{read}_T()$ does not write to the shared memory. Consequently, the implementation satisfies the read invisibility property.

A remark on $(abort, 1)$ If $(abort, 1)$ is returned to a transaction T, this is because T is executing an operation $X.\mathsf{read}_T()$, and the abortion is due to the fact that, while the values previously read by T belong to a consistent global state (“consistent

```
operation begin_T() is
(1)  $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$; $t\_depend_T \leftarrow p\_depend_i$; return()
end operation.

operation $X$.read_T() is
(2)  if ($\nexists$ local copy $lc(XX)$ of $XX$) then
(3)     allocate local space $lc(XX)$; $lc(XX) \leftarrow XX$;
(4)     $lrs_T \leftarrow lrs_T \cup \{X\}$; $t\_depend_T[X] \leftarrow lc(XX).depend[X]$;
(5)     if ($\exists Y \in lrs_T : t\_depend_T[Y] < lc(XX).depend[Y]$) then return($abort, 1$) end if;
(6)     for each $Y \notin lrs_T$ do
(7)          $t\_depend_T[Y] \leftarrow \max(t\_depend_T[Y], lc(XX).depend[Y])$
(8)     end for
(9)  end if;
(10) return($lc(XX).value$)
end operation.

operation $X$.write_T($v$) is
(11) if ($\nexists$ local copy $lc(XX)$ of $XX$) then allocate local space $lc(XX)$ end if;
(12) $lc(XX).value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$;
(13) return($ok$).
end operation.

operation try_to_commit_T() is
(14) lock all the objects in $lrs_T \cup lws_T$;
(15) if ($lrs_T \neq \emptyset$) $\wedge$ ($\exists\, Z \in lrs_T : t\_depend_T[Z] \neq Z.depend[Z]$)
(16)    then release all the locks; return($abort, 2$)
(17) end if;
(18) if ($lws_T \neq \emptyset$) then
(19)    for each $X \in lws_T$ do $t\_depend_T[X] \leftarrow XX.depend[X] + 1$ end for;
(20)    for each $X \in lws_T$ do $XX \leftarrow (lc(XX).value, t\_depend_T)$ end for
(21) end if;
(22) release all the locks;
(23) $p\_depend_i \leftarrow t\_depend_T$;
(24) return($commit$)
end operation.
```

Fig. 10.12 An STM system guaranteeing the virtual world consistency condition

snapshot"), the addition of the value of X obtained from the shared memory would make this snapshot inconsistent.

The operation X.write$_T(v)$ The algorithm implementing this operation is very simple. If there is no local copy $lc(XX)$ of the implementation register XX associated with X, one is created (line 11). Then, the value v is written into $lc(XX).value$ and the control variable lws_T is updated (line 12).

The operation try_to_commit$_T()$ When a process p_i executes try_to_commit$_T()$, it first locks all the registers accessed by T (line 14); those are the application registers whose names have been saved in $lrs_T \cup lws_T$. This locking is done according to a canonical order (e.g., on the register names) to prevent deadlock and starvation. If it is a read-only transaction (that has read more than one application register), it can be committed if its incremental snapshot is still valid, i.e., the values it has read from the shared memory have not yet been overwritten. This is what is captured by the predicate

$$\forall Z \in lrs_T : t\_depend_T[Z] = Z.depend[Z]$$

whose negation is used at line 15. If this predicate is false, the transaction T is aborted after it has released all its locks. If this predicate is false, the transaction can appear as if both its reads and its writes (if any) have been simultaneously executed just before the test of line 15 was evaluated.

If the transaction T is a write-only transaction (i.e., $lrs_T = \emptyset$, line 15), it follows from the locks on the application registers of lws_T that the transaction T can write new values of the registers in lws_T with the associated data dependencies (captured in t_depend_T) into the shared memory (line 20). Due to the fact that all the registers in lws_T are locked when they are written, it follows that these writes appear as being executed simultaneously. Before executing these writes, T has to update the sequence number of each of these registers X for the dependency vectors to have correct values (line 19).

If the transaction T is neither read-only nor write-only it can be committed only if all its read and write operations could have been executed simultaneously. As we have seen, this is ensured by the net effect of the predicate used at line 15 for the simultaneity of the reads and the use of locks on all application registers in $lrs_T \cup lws_T$.

Let us finally observe that, when a transaction returns the control value *commit* (line 24), the dependency vector of the associated process p_i has to be updated accordingly (line 23) to take into account the new data dependencies created by the newly committed transaction T.

A remark on $(abort, 2)$ If $(abort, 2)$ is returned to a read-only transaction T, the values it has incrementally read define a consistent snapshot, but this snapshot cannot be totally ordered (with certainty) with respect to the committed transactions. In that case, all the read operations issued by the aborted transaction T belong to its read prefix, and this read prefix is consistent with respect to the causal past of T.

A remark on write-only transactions and independent transactions As a write-only transaction T is such that $lrs_T = \emptyset$, it is easy to see that, due the fact that a transaction that executes $\text{try\_to\_commit}_T()$ can be aborted only at line 16, write-only transactions cannot be aborted.

Two transactions $T1$ and $T2$ are independent if $(lrs_{T1} \cup lws_{T1}) \cap (lrs_{T2} \cup lws_{T2}) = \emptyset$. It follows from the code of $\text{try\_to\_commit}_T()$ that independent concurrent transactions can commit independently.

A remark on read-only transactions A simple modification of the previous algorithms provides the following additional property: a read-only transaction T that reads a single object X is never aborted. T is then only made up of $X.\text{read}_T()$, and this operation is implemented as follows:

```
if (there is no local copy of $XX$) then
    allocate local space $lc(XX)$ for a local copy of $XX$; $lc(XX) \leftarrow XX$
end if;
return($lc(XX).value$).
```

When the implementation registers are not atomic When the implementation MWMR registers are not atomic, there is a very simple way to enrich the STM of Fig. 10.12 so that it works correctly; namely, using the locks associated with the application registers, it is sufficient to replace the statement "$lc(XX) \leftarrow XX$" at line 3 by "lock X; $lc(XX) \leftarrow XX$; unlock X".

Depending on implementation choices, concurrent transactions that try to access X when it is locked could either abort or wait. A read operation does not modify the value and holds the lock only for a short time. So, if X is locked because of a read operation, it could be beneficial to let the transaction wait instead of aborting it.

10.6 Summary

This chapter has presented the notion of a software transactional memory (STM). An STM system provides the programmer of a multiprocess program with the concept of an atomic procedure (called a transaction). The programmer has then to focus his effort on which parts of processes have to appear as being executed atomically and not on the way synchronization is implemented.

Two consistency conditions for STM systems have been introduced: opacity and virtual world consistency. Several STM systems which guarantee these conditions have been presented.

10.7 Bibliographic Notes

- The concept of a transactional memory was introduced by M. Herlihy and J.E.B. Moss in 1993 [144]. It was then investigated in a software-only context by N. Shavit and D. Touitou [255].
- The reader will find introductory motivations for STM systems in [98, 134, 140, 195]. [124] is a book entirely devoted to the theory of transactional memory systems.
- Issues related to the specification of STM systems are addressed in [252].
- Conflict detection and contention management in the context of STM systems are addressed in several papers (e.g., [122, 249, 259]).
- The opacity consistency condition is due to R. Guerraoui and M. Kapałka [123]. This definition borrows and extends notions developed in the context of serializability theory [50, 51, 221, 222].

- Lots of algorithms implementing an STM system satisfying opacity have been proposed (e.g., [42, 61, 87, 141, 155, 156, 244, 245]).
- The TL2 system, based on a global logical clock and locks, presented in Sect. 10.3 is due to D. Dice, O. Shalev, and N. Shavit [87]. The presentation given in Sect.10.3 is a simplified version of TL2. An extension of TL2 where the centralized global clock is replaced by a distributed clock is presented in [42].
- The JVSTM system based on a multi-version technique presented in Sect. 10.4 is due to J. Cachopo and A. Rito-Silva [61]. As for TL2, the presentation given in Sect. 10.4 is a simplified version of JVSTM. More details and a proof will be found in [61].
- The management and maintenance of multi-version STM systems is addressed in [223].
- The virtual world consistency condition is due to D. Imbs, J.-R. Mendivil, and M. Raynal [153, 163]. A formal definition is presented in [163]. This consistency condition can be seen as a version of opacity weakened with notions of causality developed in shared memory systems or message-passing systems (e.g., in [17, 45, 136, 186, 202, 241, 251]).
- The STM system satisfying the virtual world consistency condition described in Sect. 10.5.2 is due to D. Imbs and M. Raynal [163]. A proof of correctness, based on a formal definition of virtual world consistency can be found in that paper.
- The notion of permissiveness of an STM system was introduced in [121]. Intuitively it requires that no transaction be aborted when there is no conflict. This notion was explored in [38, 82].
- A notion of a universal construction suited to STM systems is addressed in [83, 273].

10.8 Exercises and Problems

1. The logical clock used in the TL2 STM system is a global clock, which can constitute a bottleneck in heavy load.

   Modify the algorithms in order to replace this centralized clock by a vector clock-like distributed clock.

   Solution in [42].
2. Add to the simplified version of JVSTM a garbage collection mechanism that recycles the versions of the implementation registers which are too old to be used by transactions.

   Solution in [61].
3. Let us consider that, in the STM system presented in Sect. 10.5.2 (suited to virtual world consistency), the implementation registers *XX* are MWMR regular registers

instead of being atomic registers. Let us observe that, due to the locks used in try_to_commit$_T$() (locking at line 14, and unlocking at line 16 or 22), no two processes can write to the same register concurrently, from which it follows that all the writes into an implementation register XX are sequential.

Modify the algorithms described in Fig. 10.12 to obtain an STM system that works with such implementation registers. These modifications require only to:

- Add the statement "**if** $predicate$ **then** return($abort$, 3) **end if**" between the two statements of line 22 where $predicate$ has to be appropriately defined, and
- Modify the second predicate used at line 15.

Prove then that these modifications guarantee that the implementation registers behave as if they were atomic.

Solution in [163].

Part V
On the Foundations Side: From Safe Bits to Atomic Registers

This part of the book is on the construction of atomic multi-valued registers from safe bits (binary registers). It consists of three chapters. The first chapter starts with a reminder of the definitions of safe, regular, and atomic registers (introduced in Chap. 2), and then presents various constructions of “high-level” registers (regular or atomic, respectively) from “low-level” registers (safe or regular, respectively). The second chapter presents a bounded construction of a one-bit single-writer/single-reader (SWSR) atomic register from three one-bit SWSR safe registers. Finally, the last chapter presents two approaches which allow for the construction of multi-valued multi-writer/multi-reader (MWMR) atomic registers from safe bits.

Chapter 11
Safe, Regular, and Atomic Read/Write Registers

For self-containment, this chapter starts with a short presentation of the notions of safe, regular, and atomic read/write registers (which were introduced in Chap. 2). It then presents simple wait-free implementations of "high-level" registers from "low-level" registers. The notions of "high-level" and "low-level" used here are not related to the computability power but to the abstraction level. This is because, as we will see in the next two chapters, while a regular register is easier to use than a safe register and an atomic register is easier to use than a regular register, they are all computationally equivalent; i.e., any of them can be built wait-free from any other without enriching the underlying system with additional computational power.

The proofs of the theorems stated in this chapter use the definitions and terminology introduced in Chap. 4.

Keywords Atomic/regular/safe register · Binary versus multi-valued register · Bounded versus unbounded construction · Register construction · Sequence number · SWSR/SWMR/MWSR/MWMR · Single-reader versus multi-reader · Single-writer versus multi-writer · Timestamp

11.1 Safe, Regular, and Atomic Registers

11.1.1 Reminder: The Many Faces of a Register

As just indicated, the following definitions have already been stated in Sect. 2.3.1.

Notation As far as its interface is concerned, as seen before, a register R provides the processes with a write operation denoted $R.\mathsf{write}(v)$ (or $R \leftarrow v$), where v is the value that is written, and a read operation $R.\mathsf{read}()$ (or $local \leftarrow R$, where $local$ is a local variable of the invoking process). We also use the notation $R.\mathsf{read}() \rightarrow v$ to indicate that the corresponding read of R returns the value v. Safe, regular, and atomic registers differ in the value returned by a read operation invoked in the presence of concurrent write operations.

The capacity of a register At a given time a register contains a single value, but according to the write operations issued by the processes, a register can contain distinct values at different times. So, the first dimension associated with a register is related to its size, i.e., its capacity to contain more or less information.

The simplest type of register is the *binary* register, which can store a single bit: 0 or 1. Otherwise, a register is *multi-valued*. A multi-valued register can be bounded or unbounded. A *bounded* register is one whose value domain includes b distinct values (e.g., the values from 0 up to $b-1$), where b is a constant known by the processes. Otherwise the register is unbounded. A register that can contain b distinct values is said to be b-*valued*. Its binary representation requires $B = \lceil \log_2 b \rceil$ bits. Its unary representation is more expensive as it requires b bits (the value v being then represented by a bit equal to 1 followed by $v-1$ bits equal to 0).

Access to a register This dimension concerns the number of processes that can read or write the register. As seen in Chap. 2, a register can be single- or multi-reader, and single- or multi-writer, hence the notation XWYR, where both X and Y stand for M(ulti) or S(ingle).

A register in the face of concurrency A fundamental question concerns the behavior of a register when it is concurrently accessed by several processes. Three types of registers can be distinguished.

SWMR safe register A SWMR safe register is a register whose read operation satisfies the following properties:

- A read that is not concurrent with a write operation (i.e., their executions do not overlap) returns the current value of the register.
- A read that is concurrent with one (or several) write operation(s) (i.e., their executions do overlap) returns *any* value that the register can contain.

It is important to see that, in the presence of concurrent write operations, a read can return a value that has never been written. The returned value only has to belong to the register domain. As an example, let the domain of a safe register R be $\{0, 1, 2, 3\}$. Assuming that $R = 0$, let $R.\mathsf{write}(2)$ be concurrent with a read operation. This read can return either 0 or 1 or 2 or 3. It cannot return 4, as this value is not in the domain of R, but can return 3 which has never been written.

A binary safe register can be seen as modeling a flickering bit. Whatever its previous value, the value of the register can flicker during a write operation and stabilizes to its final value only when the write finishes. Hence, a read that overlaps with a write can arbitrarily return either 0 or 1.

SWMR regular register A SWMR regular register is a SWMR safe register that satisfies the following property. This property addresses read operations in the presence of concurrency. It replaces the second item of the definition of a safe register.

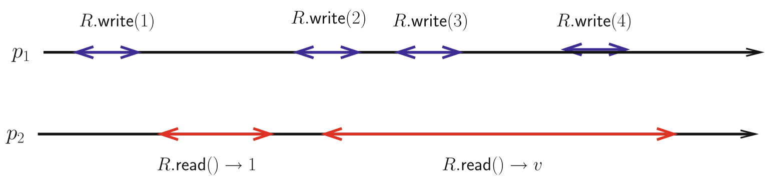


Fig. 11.1 An execution of a regular register

- A read that is concurrent with one or several write operations (i.e., the read invocation overlaps with a write invocation or with consecutive write invocations) returns the value of the register before these writes, or the value written by any of them.

An example of a regular register R (whose domain is the set $\{0, 1, 2, 3, 4\}$) written by a process p_1 and read by a process p_2 is described in Fig. 11.1. As there is no concurrent write during the first read by p_2, this read operation returns the current value of the register R, namely 1. The second read operation is concurrent with three write operations. It can consequently return any value in $\{1, 2, 3, 4\}$. If the register were only safe, this second read could return any value in $\{0, 1, 2, 3, 4\}$.

Atomic register An atomic MWMR register is such that all its operation invocations can be totally ordered in such a way that (a) each invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation. Said differently, the execution of an atomic register is linearizable (see Chap. 4).

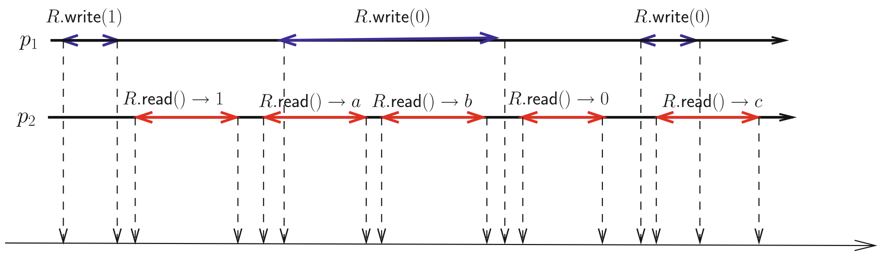
Due to the total order on all its operations and the fact that it can have several writers, an atomic register is more constrained than a regular register.

Example To illustrate the differences between safe, regular, and atomic, let us consider Fig. 11.2 (which has already been partly presented in Chap. 2). This figure considers an execution of a binary register R and presents the associated history $\widehat{H}$ (at the base level defined by the start and end events of each operation invocation).

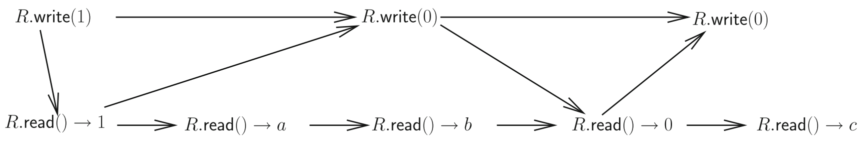
The first and third read by p_2 are issued in a concurrency-free context. Hence, whatever the type of the register (safe, regular, or atomic), the value returned is the current value of the register R. More generally, Table 11.1 describes the values returned by the read operations when the register is safe, regular, and atomic.

Let us consider the invocations of the read operation by p_2 which return the values a, b, or c.

- If R is safe, as these read invocations are concurrent with a write invocation, they can return any value (i.e., 0 or 1 as the register is binary). This is denoted $0/1$ in Table 11.1.
- If R is regular, each of the values a and b returned by the read invocation can be 1 (the value of R before the read invocation) or 0. This is because this read invocation is concurrent with a write. Differently, the value c returned by the last



History $\widehat{H}$ at the level of the start/end events associated with the operation invocations



Partial order on the read and write operations defined by $\widehat{H}$

Fig. 11.2 An execution of a register

Table 11.1 Values returned by safe, regular, and atomic registers (again)

Value returned	a	b	c
Safe	1/0	1/0	1/0
Regular	1/0	1/0	0
Atomic	1	1/0	0
Atomic	0	0	0

read invocation can only be 0 (because the value which is written concurrently is the same as the previous value of R).

- If R is atomic, there are only three possible executions, each corresponding to a correct total order on the read and write operations (as indicated above, "correct" means that the sequence of read and write invocations respects their real-time order and is such that each read invocation returns the value written by the immediately preceding write invocation).

11.1.2 From Regularity to Atomicity: A Theorem

New/old inversion It is interesting to notice that, when we look at the execution example depicted in Fig. 11.2, the only difference between a regular execution and an atomic execution is the fact that the execution with $a = 0$ and $b = 1$ is correct with respect to regularity while it is incorrect with respect to atomicity. In that execution, when we consider the two consecutive read invocations R.read() → a followed by

$R.\mathsf{read}() \to b$, the first obtains the "new" value ($a = 0$) while the "second" obtains the "old" value ($b = 1$). This is called a *new/old inversion*. This is actually *the* difference between regularity and atomicity: an atomic register is a regular register that does not allow for new/old inversions. This important observation is formalized by the theorem that follows.

Associating a write invocation with each read invocation Considering the formal notations introduced in Chap. 4, let $\widehat{H}$ be the execution history of a SWMR regular register. Let us recall that $\to_H$ is an irreflexive partial order relation on the set of the read and write invocations issued during that execution. Let op and op' be two operation invocations. $\mathsf{op} \to_H \mathsf{op}'$ if $resp[\mathsf{op}]$ occurs before $inv[\mathsf{op}']$ (i.e., $resp[\mathsf{op}] <_H inv[\mathsf{op}']$ in the associated event-based history). Let us assume without loss of generality that all the write invocations write distinct values, and let $\pi(\mathsf{r})$ be the write operation that wrote the value obtained by the read invocation r (for example, in Fig. 11.2, the first read operation r obtains the value 1, and consequently $\pi(\mathsf{r})$ is the first write operation issued by the writer).

Theorem 43 *An SWMR atomic register is an SWMR regular register such that any of its execution history $\widehat{H}$ satisfies the following property, where* $\mathsf{r1}$ *and* $\mathsf{r2}$ *are any two read invocations:* $(\mathsf{r1} \to_H \mathsf{r2}) \Rightarrow \neg\big(\pi(\mathsf{r2}) \to_H \pi(\mathsf{r1})\big)$.

This theorem states that a SWMR regular register without new/old inversion is atomic. Looking again at Fig. 11.2, as $R.\mathsf{read}() \to a \to_H R.\mathsf{read}() \to b$ and $R.\mathsf{write}(1) \to_H R.\mathsf{write}(0)$, it is not possible to have $\pi\big(R.\mathsf{read}() \to b\big) = R.\mathsf{write}(1)$ and $\pi\big(R.\mathsf{read}() \to a\big) = R.\mathsf{write}(0)$ if the execution is atomic. This theorem is particularly useful to show that a construction provides atomic registers. This is done as follows. The atomicity proof consists in incrementally proving first that the register is safe, then that it is regular, and finally that it does not allow for new/old inversion. This last point completes then the atomicity proof.

Proof The fact that a SWMR atomic register is regular and satisfies the "no new/old inversion" property is an immediate consequence of the definition of atomicity (see Chap. 4); more precisely, any execution $\widehat{H}$ of an atomic register is equivalent to a legal sequential history $\widehat{S}$ that respects the partial order $\to_H$ on its operations: two successive read operations that overlap with the same write cannot obtain first the new value and then the old value (otherwise, the witness sequential history would not be legal).

So, we only have to show the other direction, namely that a regular register whose executions have no new/old inversion is atomic. Let us first observe that, as there is a single writer, all the write operations are totally ordered. Let $\xrightarrow{w}$ be this total order relation. Moreover, let us associate with each write operation w a sequence number $sn(\mathsf{w})$, the first write being numbered 1, etc. For consistency, we also assume that there is an initial write invocation before any other read or write invocation (or equivalently, that the register has an initial value whose sequence number is 1). Let us also associate with each read operation r the sequence number $sn(\pi(\mathsf{r}))$, i.e., the sequence number of the write invocation $\pi(r)$ that wrote the value read by r.

Let us observe that, due to the fact that the register is regular and there is no new/old inversion, we have, for any two read invocations r1 and r2, $(\mathsf{r1} \rightarrow_H \mathsf{r2}) \Rightarrow \big(sn(\pi(\mathsf{r1})) \le sn(\pi(\mathsf{r2}))\big)$.

We now show that, for any execution $\widehat{H}$, there is a total order $\widehat{S}$ that is equivalent to $\widehat{H}$ (i.e., that includes the same operation invocations), is legal, and respects the partial order on the operation invocations defined by $\widehat{H}$. $\widehat{S}$ is built as follows. We start from the total order on the write operations $\stackrel{w}{\rightarrow}$ (that is included in $\widehat{H}$) and insert the read operations as follows:

- A read operation r is inserted just after the associated write operation $\pi(\mathsf{r})$.
- If two read operations r1 and r2 are such that $sn(\mathsf{r1}) = sn(\mathsf{r2})$, then insert first the one that starts first in $\widehat{H}$. (Let us recall that, at the level of the invocation/response events, any two events are totally ordered.)

Due to its very construction, $\widehat{S}$ includes all the operation invocations of $\widehat{H}$, from which it follows that $\widehat{S}$ and $\widehat{H}$ are equivalent. $\widehat{S}$ is trivially a total order (as all the operations are ordered according to their associated sequence numbers). Moreover, this total order is an extension of $\widehat{H}$, as it only adds an order on operations that are concurrent in $\widehat{H}$ (when there is no read operation concurrent with a write operation, there is no difference between regularity and atomicity). Finally, $\widehat{S}$ is legal, as each read obtains the last written value that precedes it in this total order. Hence, the corresponding history is linearizable. As this reasoning does not depend on a particular history $\widehat{H}$, it follows that the register is atomic. $\square$

As atomicity (linearizability) is a local property (Theorem 14, Chap. 4), it follows that a set of SWMR regular registers behave atomically as soon as each of them—independently from the others—satisfies the "no new/old inversion" property.

11.1.3 A Fundamental Problem: The Construction of Registers

As announced previously, a fundamental problem is the construction of high-level registers from low-level registers, the ultimate aim being the construction of a multi-valued MWMR atomic register (that can contain any value from a predefined set) from SWSR safe bits. The notion of high-level versus low-level is considered with respect to the following hierarchy. The safe registers are at the lowest level of this abstraction level hierarchy, then there the regular registers, and finally the atomic registers, which are at the top of the hierarchy.

A given construction is characterized by the following items:

- Its base registers are safe (or regular), while the high-level register that is built is regular (or atomic).
- The base registers are SWSR (or MWSR), while the constructed register is SWMR (or MWSR) or MWMR.

- The number of base registers needed to build the high-level register is a constant or depends on the number of processes. (Actually, a base register usually acts as a "copy" of the register under construction.)
- The additional control information used to build the high-level register is bounded or unbounded. Basically, unbounded means that the construction uses sequence numbers that can grow arbitrarily. Except for a few constructions, bounded constructions are much more difficult to design and prove correct than unbounded solutions. From a complexity (cost) point of view, they are always better.

11.2 Two Very Simple Bounded Constructions

This section presents two very simple bounded constructions. It focuses on safe and regular registers (let us recall that these registers have been defined as having a single writer). The first construction extends such a register from one reader to multi-reader. The second shows how to transform a safe bit into a regular bit.

11.2.1 Safe/Regular Registers: From Single-Reader to Multi-Reader

The aim of this construction is to provide an SWMR safe (or regular) register from SWSR safe (regular) registers. So, the added value here is to allow for any number of readers instead of a single reader.

The construction, described in Fig. 11.3, is very simple. The constructed SWMR register *R* is built from *n* SWSR base registers *REG*[1 : *n*], one per reader process. A reader p_i reads the base register *REG*[*i*] it is associated with, while the single writer writes all the base registers (in any order). It is important to see that this construction is bounded: it uses no additional control information, and each base register has to be of the same size (measured in number of bits) as the register we want to build. Interestingly, this construction is independent of the fact that the base registers are safe or regular.

```
operation $R$.write($v$) is
    for each $j$ in $\{1,\ldots,n\}$ do $REG[i] \leftarrow v$ end for;
    return()
end operation.

operation $R$.read() is
    return($REG[i]$)
end operation.
```

Fig. 11.3 From SWSR safe/regular to SWMR safe/regular: a bounded construction (code for p_i)

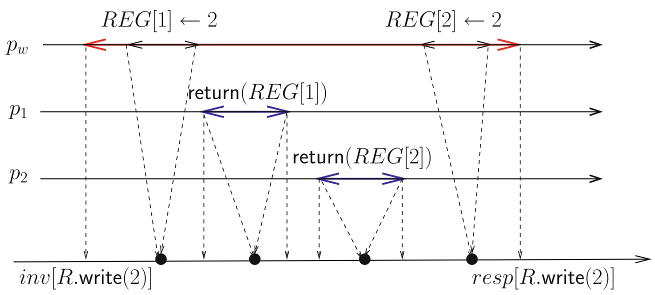


Fig. 11.4 A first counter-example to atomicity

Theorem 44 *Given one base safe (regular) register per reader, the algorithm described in Fig. 11.3 constructs an SWMR safe (regular) register.*

Proof Let us first consider safe base registers. It follows directly from the algorithm that a read of R which is not concurrent with an invocation of R.write() obtains the last value deposited in the register R. The register R is consequently safe.

Let us now consider the case where the base registers are regular. As a regular register is safe, it only remains to show that an invocation of R.read() by a process p_i that is concurrent with one or more invocations of the write operation R.write(v), R.write(v'), etc., returns one of the values v, v', etc. written by these concurrent write invocations, or the value of R before these write invocations. As $REG[i]$ is regular, it follows that, when p_i reads $REG[i]$, it obtains the value of a concurrent write on this base register (if any) or the value of $REG[i]$ before these concurrent write operations. It follows that the constructed register R is regular. □

It is important to see that unfortunately the construction of Fig. 11.3 does not build a SWMR atomic register when every base register $REG[i]$ is a SWSR atomic register. To show this, let us consider the counter-example described in Fig. 11.4, where there are one writer p_w and two readers p_1 and p_2. Let us assume that the register R contains initially the value 1 (which means that we initially have $REG[1] = REG[2] = 1$). To write the value 2 in R, the writer first executes $REG[1] \leftarrow 2$ and then $REG[2] \leftarrow 2$. The duration between these two write invocations on base registers can be arbitrary (recall that, as the processes are asynchronous, there is no assumption on their speed). Concurrently, p_1 reads $REG[1]$ and returns 2, while later (as indicated in the figure) p_2 reads $REG[2]$ and returns 1. The linearization order on the two base atomic registers is depicted in the figure (bold dots). The reader can easily see that, from the point of view of the constructed register R, there is a new/old inversion, as p_1 reads first and obtains the new value, while p_2 reads later and obtains the old value. The constructed register is consequently not atomic.

11.2.2 Binary Multi-Reader Registers: From Safe to Regular

The aim here is to build a regular bit from a safe bit. As (a) a bit has only one out of two possible values (0 or 1) and (b) regularity allows for new/old inversions when several read invocations are concurrent with one or more write invocations, the only problem that has to be solved is the following. Let us assume that the register has the value 0 and there is an invocation of the operation R.write() that writes the very same value 0. As a base register is only safe, it is possible that a concurrent invocation of R.read() obtains the value 1 (that has maybe never been written into the register). An easy way to fix this problem is to force invocations of R.write() to always write a value different from the previous one. It then follows that invocations of R.read(), which are concurrent with one or more invocations of R.write(), obtain the value before these write invocations or the value written by one of these invocations.

The corresponding construction is described in Fig. 11.5. It only requires that the (single) writer uses a local register $prev\_val$ that contains the previous value that it wrote in the base safe register *REG*. The test guarantees that a value is written in the safe base register only when it is different from its current value.

Theorem 45 *Given an SWMR binary safe register, the construction described in Fig. 11.5 builds an SWMR binary regular register.*

Proof The proof is an immediate consequence of the following facts: (1) As the underlying register is safe, a read that is not concurrent with a write obtains the last written value; (2) As the underlying safe register always alternates between 0 and 1, a read invocation concurrent with one or more write invocations obtains the value of the base register before these write invocations or one of the values written by such a write invocation. □

Remark The previous construction exploits the fact that the constructed register R can only contain one out of two possible values. Unfortunately, it cannot be extended to work for multi-valued registers, nor to implement an atomic binary register.

```
operation $R$.write($v$) is
    if ($prev\_val \neq v$) then $REG \leftarrow v$; $prev\_val \leftarrow v$ end if;
    return()
end operation.

operation $R$.read() is
    return($REG$)
end operation.
```

Fig. 11.5 SWMR binary register: from safe to regular

11.3 From Bits to b-Valued Registers

This section presents bounded constructions from bits to registers whose value domain is made up of b distinct values ($b > 2$). The base bits and the constructed registers are SWMR registers. (Of course, the algorithms trivially work when the base bits are SWSR, the constructed b-valued register being then SWSR.) Finally, the abstraction level of the constructed register is the same as the one of the base bits from which it is built: if the base bits are safe (regular or atomic) the b-valued constructed register is safe (regular or atomic).

11.3.1 From Safe Bits to b-Valued Safe Registers

Principles of the construction The construction is due to Lamport (1986). It assumes that b is an integer power of 2, $b = 2^B$, where B is an integer. It follows that (with a possible pre-encoding if the b distinct values are not the consecutive values from 0 up to $b - 1$) the binary representation of the b-valued register R that we want to build consists of exactly B bits. This means that any combination of B bits defines a value that belongs to the domain of R (notice that this would not be true if b were not an integer power of 2).

Using this encoding of the value of R, the construction is very simple. It consists in using an array $REG[1..B]$ of SWMR safe bit registers to store the current value of R. Let $\mu_i = REG[i]$. The binary representation of the current value of R is $\mu_1 \dots \mu_B$. The corresponding construction is given in Fig. 11.6.

Cost As $B = \log_2(b)$, the memory cost (space complexity) of the construction is logarithmic with respect to the size of the value domain of the constructed register R. This is an immediate consequence of the binary encoding of the values of the high-level register R.

```
operation $R$.write($v$):
  let $\mu_1 \dots \mu_B$ be the binary representation of $v$;
  for each $j$ in $\{1, \dots, B\}$ do $REG[j] \leftarrow \mu_j$ end for;
  return()
operation.

operation $R$.read() is
  for each $j$ in $\{1, \dots, B\}$ do $\mu_j \leftarrow REG[j]$ end for;
  let $v$ be the value whose binary representation is $\mu_1 \dots \mu_B$;
  return($v$)
operation.
```

Fig. 11.6 SWMR safe register: from binary domain to b-valued domain

Theorem 46 *Let $b = 2^B$. Given B SWMR safe bits, the construction described in Fig. 11.6 builds an SWMR b-valued safe register.*

Proof The proof is trivial. A read of R that does not overlap with a write of R obtains the binary representation of the last value written into R, and is consequently correct. A read of R that overlaps with a write of R can obtain any of b possible values whose binary encoding uses B bits. As all the possible combinations of the B base bit registers represent exactly the b values that R can potentially contain (this is because $b = 2^B$), it follows that a read concurrent with a write operation returns a value that belongs to the domain of R. Consequently, R is a b-valued safe register. □

Remark It is interesting to notice that this construction does not build a regular register R even when the base bit registers are regular. A read that overlaps with a write operation that changes the value of R from $0 \ldots 0$ to $1 \ldots 1$ (in binary representation) can return any value. The reader can check that this remains true if both the read and write operations access the array $REG[1 : B]$ in some predefined order.

11.3.2 From Regular Bits to Regular b-Valued Registers

Principles of the construction A way to build a SWMR regular b-valued register R from regular bits consists in employing unary encoding. Considering an array $REG[1..b]$ of SWMR regular bits, the value $v \in [1..b]$ is represented by 0s in bits 1 through $v - 1$ and 1 in the vth bit.

The construction is described in Fig. 11.7. The idea is to write into the $REG[1..b]$ array in one direction, and to read it in the opposite direction. To write v, the writer first sets $REG[v]$ to 1, and then "cleans" the array REG. The cleaning consists in setting the bits $REG[v - 1]$ up to $REG[1]$ to 0, the writing being done in the direction from $v - 1$ down to 1. To read, a reader traverses the array $REG[1..b]$ starting from its first entry ($REG[1]$) and stopping as soon as it discovers an entry j such that

```
operation $R$.write($v$) is
    $REG[v] \leftarrow 1$;
    for $j$ from $v-1$ step $-1$ until $1$ do $REG[j] \leftarrow 0$ end for;
    return()
end operation.

operation $R$.read() is
    $j \leftarrow 1$;
    while $(REG[j] = 0)$ do $j \leftarrow j+1$ end while;
    return($j$)
end operation.
```

Fig. 11.7 SWMR regular register: from binary domain to b-valued domain

$REG[j] = 1$. It then returns j as the result of the read operation. It is important to see that a read operation starts reading first the "cleaned" part of the array.

It is important to see that, even when no write operation is in progress, it is possible that several entries of the array are equal to 1. The value represented by the array is then the value v such that $REG[v] = 1$ and for all the entries $1 \le j < v$ we have $REG[j] = 0$. Those entries are then the only meaningful entries. The other entries can be seen as an old print of past values of the constructed register.

The construction assumes that the register R has an initial value, say v. The array $REG[1 : b]$ is accordingly initialized, i.e., $REG[j] = 0$ for all $j \neq v$ and $REG[v] = 1$.

It can be observed from the algorithm that implements the operation $R.\mathsf{write}()$ that, once set to 1, the "last" base register $REG[b]$ keeps that value forever. It can therefore be eliminated. The writer never writes it and, when it has to read it, a reader can correctly consider that its value is 1.

As the read algorithm does not write base registers, it works whatever the number of readers. The only constraint is on the base registers, which have to be SWMR if there are several readers (as each reader reads the base registers), and can be SWSR when there is a single reader. (This remark remains valid for the atomic construction described in Sect. 11.3.3.)

Cost The memory cost of the construction is b base bits; i.e., it is linear with respect to the size of the value domain of the constructed register R. This is a consequence of the unary encoding of these values. Let B be the number of bits required to obtain a binary representation of a value of R. It is important to see that, as $B = \log_2 b$, the cost of the construction is exponential with respect to this number of bits B.

Lemma 24 *Any invocation of* $R.\mathsf{read}()$ *or* $R.\mathsf{write}()$ *terminates. Moreover, the value* v *returned by a read belongs to the set* $\{1, \dots, b\}$.

Proof An invocation of $R.\mathsf{write}()$ trivially terminates (as by definition the **for** loop always terminates). For the termination of an invocation of $R.\mathsf{read}()$, let us first make the following two observations:

- First, let us notice that one entry of the array REG is initially equal to 1. It then follows from the write algorithm that, each time the writer changes the value of a base register $REG[x]$ from 1 to 0, it has previously set to 1 another entry $REG[y]$ such that $x < y \le b$. Consequently, if the writer updates $REG[x]$ from 1 to 0 while concurrently the reader reads $REG[x]$ and obtains the new value 0, we can conclude that a higher entry of the array has the value 1.

- The second observation is the following. If, while the previous value of $REG[x]$ was 1, the reader reads it and concurrently the writer updates $REG[x]$ to the value 1, the reader obtains the value 1, as the base register is regular (if the base register were only safe, the reader could obtain the value 0).

It follows from these observations that a sequential scanning of the array REG (starting at $REG[1]$) necessarily encounters en entry $REG[v]$ whose reading returns 1. As the running index of the **while** loop starts at 1 and is increased by 1 each time the

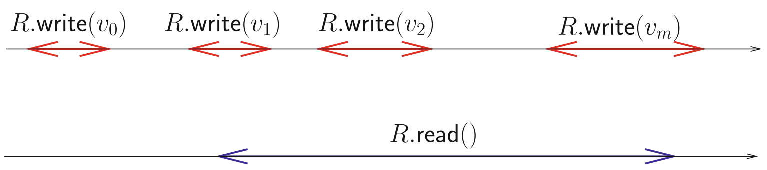


Fig. 11.8 A read invocation with concurrent write invocations

loop body is executed, it follows that the loop always terminates, and the value j it returns is such that $1 \le j \le b$. $\square$

Remark The previous lemma relies heavily on the fact that the register R can contain only b distinct values. The lemma would no longer be true if the value domain of R was unbounded. An invocation of R.read() could then never terminate in the case where the writer continuously writes increasing values. This is due to the following possible scenario. Let R.write(x) be the last write invocation terminated before the invocation R.read(), and let us assume that there is no concurrent write invocation R.write(y) such that $y < x$. It is possible that, when it reads *REG*$[x]$, the reader finds *REG*$[x] = 0$ because another R.write(y) operation (with $y > x$) updated *REG*$[x]$ from 1 to 0. Now when it reads *REG*$[y]$, the reader finds *REG*$[y] = 0$ because another R.write(z) operation (with $z > y$) updated *REG*$[y]$ from 1 and so on. The read invocation can then never terminate.

Theorem 47 *Given b SWMR regular bits, the construction described in Fig. 11.7 builds an SWMR b-valued regular register.*

Proof Let us first consider a read operation that is not concurrent with a write, and let v be the last written value. It follows from the write algorithm that, when R.write(v) terminates, the first entry of the array equal to 1 is *REG*$[v]$ (i.e., *REG*$[x] = 0$ for $1 \le x \le v - 1$). As a read scans the array starting from *REG*[1], then *REG*[2], etc., it necessarily reads until *REG*$[v]$ and returns accordingly the value v.

Let us now consider a read operation R.read() that is concurrent with one or more write operations R.write(v_1), ..., R.write(v_m) (as depicted in Fig. 11.8). Moreover, let v_0 be the value written by the last write invocation that terminated before the invocation R.read() starts (or the initial value if there is no such write invocation). As a read invocation always terminates (Lemma 24), the number of write invocations concurrent with the R.read() invocation is finite. We have to show that the value v returned by R.read() is one of the values $v_0, v_1, \ldots, v_m$. We proceed by case analysis:

1. $v < v_0$.
   No value that is both smaller than v_0 and different from v_x ($1 \le x \le m$) can be output. This is because (1) R.write(v_0) has set to 0 all the entries from $v_0 - 1$ until the first one, and only a write of a value v_x can set *REG*$[v_x]$ to 1; and (2) as the base registers are regular, if *REG*$[v']$ is updated by an invocation R.write(v_x) from 0 to the same value 0, the reader cannot concurrently read *REG*$[v'] = 1$. It follows from this observation that, if R.read() returns a value v smaller than

v_0, that value has necessarily been written by a concurrent write invocation, and consequently R.read() satisfies the regularity property.

2. $v = v_0$.
In this case, R.read() trivially satisfies the regularity property. (Let us notice that it is possible that the corresponding write invocation be some R.write(v_x) such that $v_x = v_0$.)

3. $v > v_0$.
From $v > v_0$ we can conclude that the read invocation obtained 0 when it read $REG[v_0]$. As $REG[v_0]$ was set to 1 by R.write(v_0), this means that there is an invocation R.write(v'), issued after R.write(v_0) and concurrent with R.read(), such that $v' > v_0$, and that invocation has executed $REG[v'] \leftarrow 1$ and has then set to 0 at least all the registers from $REG[v' - 1]$ up to $REG[v_0]$. We consider three cases:

   (a) $v_0 < v < v'$.
   In this case, as $REG[v]$ was set to 0 by R.write(v'), we can conclude that there is a R.write(v), issued after R.write(v') and concurrent with R.read(), that updated $REG[v]$ from 0 to 1. The value returned by R.read() is consequently a value written by a concurrent write invocation. The regularity property is consequently satisfied by R.read().

   (b) $v_0 < v = v'$.
   The regularity property is then trivially satisfied by R.read(), as R.write (v') and R.read() are concurrent.

   (c) $v_0 < v' < v$.
   In this case, R.read() missed the value 1 in $REG[v']$. This can only be due to a R.write(v'') operation, issued after R.write(v') and concurrent with R.read(), such that $v'' > v'$, and that operation has executed $REG[v''] \leftarrow 1$ and has then set to 0 at least all the registers from $REG[v'' - 1]$ to $REG[v']$. We are now in the same situation as the one described at the beginning of item 3, where v_0 and R.write(v') are replaced by v' and R.write(v''), respectively. As the number of values between v_0 and b is finite and as the invocations of R.read() terminate, if follows that it eventually terminates in case 3a or case 3b, which completes the proof of the theorem. □

A counter-example for atomicity Figure 11.9 shows that, even if all the base registers are atomic, the previous construction does not provide an atomic b-valued register.

Let us assume that $b = 5$ and the initial value of the register R is 3, which means that we initially have $REG[1] = REG[2] = 0$, $REG[3] = 1$, and $REG[4] = REG[5] = 0$. The writer issues first R.write(1) and then R.write(2). There are concurrently two read invocations as indicated in the figure. The first returns the value 2, while the second one returns the value 1. Hence, there is a new/old inversion. The last line of the figure describes a linearization order $\widehat{S}$ of the read and write invocations on

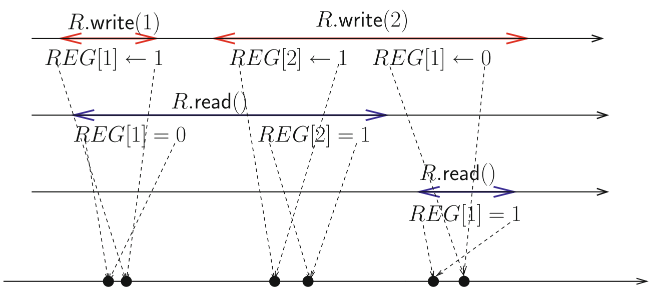


Fig. 11.9 A second counter-example for atomicity

the base binary registers. (As we can see, each base object taken alone is linearizable. This follows from the fact that linearizability is a local property; see Sect. 4.4.)

11.3.3 From Atomic Bits to Atomic b-Valued Registers

As just seen, the previous construction does not work to build a b-valued atomic register from atomic bits. Interestingly, a relatively simple modification of its read algorithm prevents new/old inversions from occurring. The construction that follows is due to K. Vidyasankar (1989).

Principles of the construction The idea consists in decomposing a R.read() operation into two phases. The first phase is the previous read algorithm: it reads the base registers in ascending order, until it finds an entry equal to 1; let j be that entry. Then, the second phase traverses the array in the reverse direction (from j to 1), and determines the smallest entry that contains the value 1 to return it. So, the returned value is determined by a double scanning of a "meaningful" part of the *REG* array.

The construction is given in Fig. 11.10. To understand the way it works let us consider the first invocation of R.read() depicted in Fig. 11.9. After it finds $REG[2] = 1$, it changes its scanning direction. It then finds $REG[1] = 1$ and returns consequently the value 1. The second read (that starts after the first one) will return the value 1 or 2 according to the value read from $REG[1]$. If it reads 1, as in the figure, the read invocation returns 1. This shows that, in the presence of concurrency, this construction does not strive to eagerly return a value. Instead, the value v returned by a read operation has to be "validated" by an appropriate procedure, namely, all the "preceding" base registers $REG[v-1]$ until $REG[1]$ have to be found equal to 0 when read for the second time.

Theorem 48 *Given b SWMR atomic bits, the construction described in Fig. 11.10 builds an SWMR atomic b-valued register.*

Proof The proof consists of two parts: first showing that the constructed register is regular, and then showing that it does not allow for new/old inversions. Applying Theorem 43 proves then that the constructed register is an SWMR atomic register.

```
operation $R$.write($v$) is
        $REG[v] \leftarrow 1$;
        for $j$ from $v-1$ step $-1$ until $1$ do $REG[j] \leftarrow 0$ end for;
        return()
end operation.

operation $R$.read() is
(1)  $j\_up \leftarrow 1$;
(2)  while $(REG[j\_up] = 0)$ do $j\_up \leftarrow j\_up + 1$ end while;
(3)  $j \leftarrow j\_up$;
(4)  for $j\_down$ from $j\_up - 1$ step $-1$ until $1$ do
(5)      if $(REG[j\_down] = 1)$ then $j \leftarrow j\_down$ end if
(6)  end for;
(7)  return($j$)
end operation.
```

Fig. 11.10 SWMR atomic register: from bits to a b-valued register

Let us first show that the constructed register is regular. Let R.read() be a read invocation and j the value it returns. We consider two cases (let us observe that, due to the construction, the case $j > j\_up$ cannot happen):

- $j = j\_up$ (j is determined at line 3).
  The value returned is then the same as the one returned by the construction described in Fig. 11.7. It follows from Theorem 47 that the value read is then either the value of the last preceding write or the value of a concurrent write invocation.
- $j < j\_up$ (j is determined at line 5).
  In this case, the read found $REG[j] = 0$ during the ascending loop (line 2) and $REG[j] = 1$ during the descending loop (line 5). Due to the atomicity of the $REG[j]$ register, this means that a write operation has written $REG[j] = 1$ between these two readings of that base atomic register. It follows that the value j returned has been written by a concurrent write operation.

To show that there is no new/old inversion, let us consider Fig. 11.11. There are two write invocations and two read invocations r1 and r2, which are concurrent with the second write invocation. (The fact that the read invocations are issued by the same process or different processes is irrelevant.) As the constructed register R is regular, both read invocations can return v or v'. If the first read invocation r1 returns v, the second read invocation r2 can return either v or v' without entailing a new/old inversion. So, let us consider the case where r1 returns v'. We show that r2 returns v'', where v'' is v' or a value written by a more recent write concurrent with this read. If $v'' = v'$, there is no new/old inversion. So, let us consider $v'' \neq v'$. As r1 returns v', r1 has sequentially read $REG[v'] = 1$ and then $REG[v'-1] = 0$ to $REG[1] = 0$ (lines 4–5). Moreover, $r2$ starts after $r1$ has terminated (r1 $\rightarrow_H$ r2) in the associated execution history $\widehat{H}$).

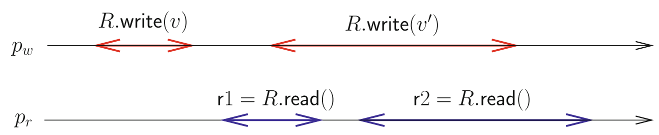


Fig. 11.11 There is no new/old inversion

1. $v'' < v'$. In this case, a write invocation has written $REG[v''] = 1$ after $r1$ has read $REG[v''] = 0$ (at line 5) and before $r2$ reads $REG[v''] = 1$ (at line 3 or line 4) with $1 \le v'' < v'$. It follows that this write invocation is after R.write(v') (there is a single sequential writer, and r1 returns v'). Consequently, r2 obtains a value more recent than v' (hence more recent than v), and there is no new/old inversion.

2. $v'' > v'$. In this case, r2 has read 1 from $REG[v'']$ and then 0 from $REG[v']$ (line 5). As r1 terminates (reading $REG[v'] = 1$ and returning v') before the invocation r2 starts and the write invocations are sequential, it follows that there is a write invocation, issued after R.write(v'), that has updated $REG[v']$ from 1 to 0.

   (a) If that operation is R.write(v''), we conclude that the value v'' read by r2 is newer than v' and there is no new/old inversion.

   (b) If that operation is not R.write(v''), it follows that there is another invocation R.write(v'''), such that $v''' > v'$, that has updated $REG[v']$ from 1 to 0, and that update was issued after R.write(v') (that set $REG[v']$ to 1) and before $r2$ reads $REG[v'] = 0$.

   Moreover, R.write(v''') is before R.write(v'') (otherwise, the update of $REG[v']$ from 1 to 0 would have been done by R.write(v'')). It follows that R.write(v''') is after R.write(v') and before R.write(v''), from which we conclude that v'' is more recent than v', proving that there is no new/old inversion. □

11.4 Three Unbounded Constructions

This section presents three constructions based on sequence numbers which, consequently, are unbounded. The sequence numbers are used to identify each write invocation and associate a total order with them, a total order that can then be easily exploited. The use of sequence numbers makes these constructions relatively simple. (It is possible to design equivalent constructions that are bounded: they use only a constant number of safe bits and are much more involved.)

The high-level register R that we want to build can be bounded or not. This depends only on the application that uses it. It is important to see that, whether R is bounded or not, the base registers from which it is built contain sequence numbers and are consequently potentially unbounded. More explicitly, in the constructions presented in this section, a base register *REG* is made up of two fields:

- *REG*.val is the data part intended to contain the value v of the constructed register R. As already noticed, whether this part is bounded or not depends only on the upper-layer application.
- *REG*.sn is a control part containing a sequence number and possibly a process identity. The sequence number values increase proportionally to the number of write invocations, and consequently cannot be bounded.

11.4.1 SWSR Registers: From Unbounded Regular to Atomic

As soon as we can use sequence numbers and those can be kept inside a base regular register, it becomes easy to build an SW1R atomic register from an unbounded regular SWSR. The underlying principle consists in associating a sequence number with each write operation and use it to prevent the new/old inversion phenomenon from occurring. It then follows from Theorem 43 that the constructed register is atomic.

The construction is described in Fig. 11.12. The local variable sn of the writer is used to generate sequence numbers. Each time it writes a value v, the writer deposits the pair $\langle sn, v\rangle$ in the base regular register *REG*. The reader manages two local variables: $last\_sn$ stores the greatest sequence number it has ever seen, and $last\_val$ stores the corresponding value. When it wants to read the high-level register R, the reader first reads the base regular register *REG*, and then compares $last\_sn$ and the sequence number it has just read in order to prevent old/new inversions. The scope of the local variable aux used by the reader spans a read invocation; it is made up of two fields: a sequence number ($aux.sn$) and a value ($aux.val$).

Theorem 49 *Given an unbounded SWSR regular register, the construction described in Fig. 11.12 builds an SWSR atomic register.*

```
operation $R$.write($v$) is
    $sn \leftarrow sn + 1$; $REG \leftarrow \langle sn, v\rangle$;
    return()
end operation.

operation $R.read()$:
    $aux \leftarrow REG$;
    if ($aux.sn > last\_sn$) then $last\_sn \leftarrow aux.sn$;
                                  $last\_val \leftarrow aux.val$
    end if;
    return($last\_val$)
end operation.
```

Fig. 11.12 SWSR register: from regular to atomic (unbounded construction)

Proof The proof is similar to the proof of Theorem 43. Let us associate with each read invocation r the sequence number (denoted $sn(\mathsf{r})$) of the value it returns as a result (this is possible as the base register is regular and consequently a read always returns a value that was written, that value being the last written value or a value concurrently written, if any). Considering an arbitrary execution history $\widehat{H}$ of the register R, we show that $\widehat{H}$ is atomic (linearizable) by building an equivalent sequential history $\widehat{S}$ that is legal and respects the partial order on the operations defined by $\rightarrow_H$.

$\widehat{S}$ is built from the sequence numbers associated with the invocations. First, let us order all write invocations according to their sequence numbers. Then, let us order each read invocation just after the write invocation that has the same sequence number. If two read invocations have the same sequence number, we order first the one which started first.

The history $\widehat{S}$ is trivially sequential as all the invocations are placed one after the other. Moreover, $\widehat{S}$ is equivalent to $\widehat{H}$ as it is made up of the same operation invocations. $\widehat{S}$ is trivially legal as each read follows the corresponding write invocation. We now show that $\widehat{S}$ respects $\rightarrow_H$.

- For any two write operations w1 and w2 we have either w1 $\rightarrow_H$ w2 or w2 $\rightarrow_H$ w1. This is because there is a single writer and it is sequential: as the variable sn is increased by 1 between two consecutive write invocations, no two write invocations have the same sequence number, and these numbers agree with the occurrence order of the write invocations. As the total order on the write invocations in $\widehat{S}$ is determined by their sequence numbers, their total order in $\widehat{H}$ follows.
- Let op1 be a write or a read invocation, and op2 be a read invocation such that op1 $\rightarrow_H$ op2. It follows from the construction that $sn(\mathsf{op1}) \leq sn(\mathsf{op2})$ (where $sn(\mathsf{op})$ is the sequence number of the invocation op). The ordering rule guarantees that op1 is ordered before op2 in $\widehat{S}$.
- Let op1 be a read invocation and op2 be a write invocation. Similarly to the previous item we then have $sn(\mathsf{op1}) < sn(\mathsf{op2})$, and consequently op1 is ordered before op2 in $\widehat{S}$ (which concludes the proof). □

One might think of a naive extension of the previous algorithm to construct an SWMR atomic register from base SWSR regular registers. Indeed, we could, at first glance, consider a construction associating one SWSR regular register per reader, and have the writer write in all of them, each reader reading its dedicated register. Unfortunately, a fast reader might see a new concurrently written value, whereas a reader that comes later sees the old value. This is because the second reader does not know about the sequence number and the value returned by the first reader. The latter stores them locally. In fact, with the previous construction, this can happen even if the base SWSR registers are atomic. The construction of SWMR atomic register from base SWSR atomic registers is addressed in the next section.

11.4.2 Atomic Registers: From Unbounded SWSR to SWMR

Section 11.2.1 presented a construction that builds a SWMR safe/regular register from similar SWSR base registers. It has also been shown that the corresponding construction does not build an SWMR atomic register when the base registers are SWSR atomic (see the counter-example presented in Fig. 11.4).

This section provides such a construction: assuming SWSR atomic registers, it shows how to go from single-reader registers to a multi-reader register. As it uses sequence numbers, this construction requires unbounded base registers. Thanks to the "comfort" provided by sequence numbers, this construction remains simple, but despite this "comfort", it is not as simple as the previous constructions.

Principle and description of the construction As there are now several readers, each write invocation issued by the writer writes a separate base SWSR atomic register per potential reader. Moreover, it helps the set of readers by associating with this value a sequence number. The code of the writer is depicted in Fig. 11.13.

To prevent new/old inversions as seen in Fig. 11.4, the read invocations have to help each other. To that end they use pairs $\langle$sequence number, value$\rangle$ created by the writer process. The array denoted $HELP[1..n, 1..n]$ is used to store these helping pairs. $HELP[i, j]$ is an SWSR atomic register that contains a pair $\langle$sequence number, value$\rangle$ as created by the writer. It is written only by p_i and read only by p_j. It is used as follows to ensure the atomicity of the SWMR register R that is built.

- Helping the others. Just before returning the value v it has determined, a reader p_i helps each process p_j by indicating to p_j the last value it has read (namely v) and its sequence number sn. This is realized by p_i which updates $HELP[i, j]$ to the pair $\langle sn, v\rangle$. This allows p_j not to return in the future a value older than v, i.e., a value whose sequence number would be smaller than sn.
- To be helped by the others. To determine the value returned by a read operation, a reader p_i first computes the greatest sequence number that it can know. This computation involves all SWSR atomic registers that p_i can read, i.e., $REG[i]$ and $HELP[j, i]$ for any j. The process p_i then returns the value that has the greatest sequence number it has computed.

The corresponding construction algorithm is described in Fig. 11.13. It is due to P. Vitányi and B. Awerbuch (1987).

The variable aux is a local array used by a reader; its jth entry is used to contain the $\langle$sequence number, value$\rangle$ pair that p_j has written in $HELP[j, i]$ in order to help p_i; $aux[j].sn$ and $aux[j].val$ denote the corresponding sequence number and the associated value, respectively. Similarly, reg is a local variable used by a reader p_i to contain the last $\langle$sequence number, value$\rangle$ pair that p_i has read from $REG[i]$ ($reg.sn$ and $reg.val$ denote the corresponding fields).

Remark The register $HELP[i, i]$ is used only by p_i, which can consequently keep its value in a local variable. This means that the SWSR atomic register $HELP[i, i]$

```
**operation** $R$.write($v$) **is**
    $sn \leftarrow sn + 1$;
    **for each** $j$ **in** $\{1, \ldots, n\}$ **do** $REG[i] \leftarrow \langle sn, v\rangle$ **end for**;
    return()
**end operation**.

**operation** $R$.read() **is**
    $reg \leftarrow REG[i]$;
    **for each** $j$ **in** $\{1, \ldots, n\}$ **do** $aux[j] \leftarrow HELP[j, i]$ **end for**;
    **let** $sn\_max$ **be** $\max(reg.sn, aux[1].sn, \ldots, aux[n].sn)$;
    **let** $val$ **be** $reg.val$ or $aux[k].val$ **such that** the associated seq number is $sn\_max$;
    **for each** $j$ **in** $\{1, \ldots, n\}$ **do** $HELP[i, j] \leftarrow \langle sn\_max, val\rangle$ **end for**;
    return($val$)
**end operation**.
```

Fig. 11.13 Atomic register: from one reader to multi-reader (unbounded construction)

can be used to contain the SWSR atomic register *REG*[i]. It follows that the protocol requires exactly n^2 base SWSR atomic registers.

Theorem 50 *Given n^2 unbounded SWSR atomic registers, the construction described in Fig. 11.13 builds an SWMR atomic register.*

Proof As for Theorem 49, the proof consists in showing that the sequence numbers allow the definition of a linearization of any execution history $\widehat{H}$.

Considering an execution history $\widehat{H}$ of the constructed register R, we first build an equivalent sequential history $\widehat{S}$ by ordering all the write invocations according to their sequence numbers, and then inserting the read invocations as in Theorem 49. This history is trivially legal as each read invocation is ordered just after the write operation that wrote the value that is read. Finally, a reasoning similar to the one used in Theorem 49 but based on the sequence numbers provided by the arrays *REG*[1..n] and $HELP[1..n, 1..n]$ shows that $\widehat{S}$ respects $\rightarrow_H$. □

11.4.3 Atomic Registers: From Unbounded SWMR to MWMR

This section shows how to use sequence numbers to build an MWMR atomic register from n SWMR atomic registers (where n is the number of writers). The construction is simpler than the previous one. An array *REG*[1..n] of n SWMR atomic registers is used in such a way that p_i is the only process that can write *REG*[i], while any process can read it. Each register *REG*[i] stores a ⟨sequence number, value⟩ pair. As before $X.sn$ and $X.val$ are used to denote the sequence number field and the value field of the register X, respectively. Each *REG*[i] is initialized to the same pair, namely ⟨0, v_0⟩, where v_0 is the initial value of R.

```
operation $R$.write($v$) is
    for each $j$ in $\{1, \ldots, n\}$ do $reg[j] \leftarrow REG[j]$ end for;
    let $sn\_max$ be $\max(reg[1].sn, \ldots, reg[n].sn) + 1$;
    $REG[i] \leftarrow \langle sn\_max, v\rangle$;
    return()
end operation.

operation $R$.read() is
    for each $j$ in $\{1, \ldots, n\}$ do $reg[j] \leftarrow REG[j]$ end for;
    let $k$ be the process identity such that $[sn, k]$ is the greatest timestamp
        among the $n$ timestamps $\langle reg[1].sn, 1\rangle$, ... and $\langle reg[n].sn, n\rangle$;
    return($reg[k].val$)
end operation.
```

Fig. 11.14 Atomic register: from one writer to multi-writer (unbounded construction)

The problem to solve consists in allowing the writers to totally order their write operations. To that end, the idea is the following. A write invocation first computes the highest sequence number that was used, say sn, and defines $sn + 1$ as the next sequence number. Unfortunately, this does not prevent two distinct write invocations from associating the same sequence number with their respective values. A simple way to cope with this problem consists in associating a *timestamp* with each value, where a timestamp is a pair made up of a sequence number plus the identity of the process that issues the corresponding write invocation.

The timestamping mechanism can be used to define a total order on all the timestamps as follows. Let $ts1 = \langle sn1, i\rangle$ and $ts1 = \langle sn2, j\rangle$ be any two timestamps. We have

$$ts1 < ts2 \overset{\text{def}}{=} \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

The corresponding construction is described in Fig. 11.14. The meaning of the additional local variables that are used is clear from the context (and from the similar variables used in the previous constructions).

Theorem 51 *Given n unbounded SWMR atomic registers, the construction described in Fig. 11.14 builds an MWMR atomic register.*

Proof As previously, the proof consists in showing that the timestamps allow the definition of a linearization of any execution history $\widehat{H}$.

Considering an execution history $\widehat{H}$ of the constructed register R, we first build an equivalent sequential history $\widehat{S}$ by ordering all the write invocations according to their timestamps, then inserting the read invocations as in Theorem 49. This history is trivially legal as each read invocation is ordered just after the write invocation that wrote the read value. Finally, a reasoning similar to the one used in Theorem 49, based on timestamps, shows that $\widehat{S}$ respects $\rightarrow_H$. $\square$

11.5 Summary

Considering (a) the abstraction level hierarchy associated with registers as defined by safe, regular, and atomic registers, (b) the fact that a register can be SWSR, SWMR, MWSR, or MWMR, and (c) the size of the registers (binary versus multi-valued), this chapter has presented bounded and unbounded constructions from one type of read/write register to another type of read/write register. More precisely, the following constructions have been presented:

- Bounded constructions:
  - From *safe* (*regular*) SWSR registers to *safe* (*regular*) SWMR registers (Sect. 11.2.1)
  - From *safe* binary SWMR registers to *regular* binary SWMR registers (Sect. 11.2.2)
  - From *binary* safe SWMR registers to *multi-valued* safe SWMR registers (Sect. 11.3.1)
  - From *binary* regular SWMR registers to *multi-valued* regular SWMR registers (Sect. 11.3.2)
  - From *binary* atomic SWMR registers to *multi-valued* atomic SWMR registers (Sect. 11.3.3).
- Unbounded constructions:
  - From *regular* SWSR registers to unbounded *atomic* SWSR registers (Sect. 11.4.1)
  - From atomic SWSR registers to unbounded atomic SWMR registers (Sect. 11.4.2)
  - From atomic SWMR registers to unbounded atomic MWMR registers (Sect. 11.4.3)

11.6 Bibliographic Notes

- The notions of safe, regular, and atomic registers are due to L. Lamport [189].

  Constructions from one type of register into another one were proposed by L. Lamport [190]. Very interestingly, L. Lamport presented in [190] a stacking of wait-free constructions which allows a b-valued atomic MWMR register to be built from binary SWSR safe registers. The first intuition of these types of registers can be found in [184].
- Axioms for asynchronous shared memory access are presented in [206].
- The bounded constructions described in Fig. 11.5 (from a safe bit to an atomic bit), Fig. 11.6 (from safe bits to a b-valued safe register), and Fig. 11.7 (from regular bits to a b-valued regular register) are due to L. Lamport [190].

- The bounded construction for building an atomic b-valued register from atomic bits which is described in Fig. 11.10 is due to K. Vidyasankar [268].
- The unbounded constructions from SWSR to SWMR described in Fig. 11.13 and from SWMR to MWMR described in Sect. 11.4.3 are due to P. Vitányi and B. Awerbuch [272]. This paper presents also corresponding bounded constructions, and the previous two unbounded constructions are a very first step in the design of these bounded constructions.
- Many constructions from one type of register into another one have been proposed (e.g., [22, 43, 52, 59, 132, 168, 177, 178, 196, 219, 225, 257, 265, 268, 269, 270, 271] to cite a few).
- The notion of a multi-valued regular register was also investigated in [72, 254] and the cost of multi-valued register implementations is discussed in [73].

Chapter 12
From Safe Bits to Atomic Bits: Lower Bound and Optimal Construction

Constructions of higher-abstraction-level read/write registers from lower-abstraction-level read/write registers were presented in the previous chapter. As we have seen, the notion of "abstraction level" that was considered has several dimensions, namely (a) the safety, regularity, or atomicity behavior of the base registers, (b) the number of values they can contain, and (c) the fact that they are SWSR, SWMR, or MWMR registers.

One of these constructions has shown how to build an SWSR atomic register from an SWSR safe register. This construction, which is based on sequence numbers, was particularly simple. Unfortunately, as it uses sequence numbers, it is an unbounded construction. This chapter presents a bounded construction that builds an atomic bit from SWSR safe (or regular) bits. This construction, which is due to J. Tromp (1989), is optimal with respect to the number of safe bits that are used; namely, it requires only three safe bits.

Before presenting this non-trivial construction, this chapter presents a lower bound theorem due to L. Lamport (1986). This theorem states that, in any bounded construction of an SWSR atomic register from SWSR safe (or regular) registers, both the reader and the writer must write the shared memory which means that bidirectional communication is necessary.

Keywords Atomic bit · Bounded construction · Linearizability · Safe bit · Switch · Wait-free construction

12.1 A Lower Bound Theorem

As just indicated, the construction of an SWSR atomic register from an SWSR regular register presented in Fig. 11.12 uses sequence numbers which increase forever and consequently makes this construction unbounded. (These sequence numbers are used to prevent new/old inversions.) This construction is such that (a) each invocation of

R.write() writes a pair $\langle seq\ number, value\rangle$ in the shared memory while (b) each invocation of R.read() has only to read the shared memory.

Hence, a fundamental question is the following: Is it possible to build an atomic register from a finite number of base safe (or regular) registers that can (a) contain only a bounded number of values, and (b) be written only by the writer (of the atomic register). This section shows that such a construction is impossible.

12.1.1 Two Preliminary Lemmas

The two following lemmas will be used in the impossibility proof. The first shows that it is possible to replace several regular registers by a single register while preserving regularity. The second states a property of a sub-sequence with respect to the sequence it originated from.

Lemma 25 *Let us consider a set of SWSR regular registers, all written by the same writer process and read by the same reader process. These registers can be replaced by a single SWSR regular register.*

Proof Let $REG_1, \ldots, REG_n$ be the set of n regular registers written by the writer and read by the reader. Moreover, let VAL_i be the value domain of REG_i and $v_i \in VAL_i$ be a value of REG_i.The construction is as follows:

- The set of n registers $(REG_i)_{1\le i\le n}$ is replaced by a single regular register R whose value domain is the cross-product $VAL_1 \times VAL_2 \times \ldots \times VAL_n$.
- Let us first observe that, for any register REG_i, the writer can always keep a copy of the last value it has written in that register. Assuming $REG_1 = v_1, \ldots, REG_n = v_n$, the write of the value v_i' in the register REG_i is replaced by the write of the composite value $\langle v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n\rangle$ in the regular register R.
- A read of REG_i is realized by a read of R followed by the extraction of the value of its ith field.

To show that this construction is correct, let us consider a read of a register REG_i (denoted r_i). If that read is not concurrent with an invocation of a write of a register REG_j, the value returned by r_i returns the current value of REG_i (the ith field of R) and consequently satisfies regularity.

So, let us consider the case where there are invocations of write operations $\mathsf{w}, \ldots, \mathsf{w}'$ into some registers REG_j which are concurrent with r_i. It follows from the regularity of R that the read of R implementing r_i returns the value of R before or after one of these write invocations. There are two cases:

- None of the write invocations $\mathsf{w}, \ldots, \mathsf{w}'$ writes REG_i. Due to the construction, the ith field of R is not changed by this sequence of write invocations. It follows that r_i returns the current value v_i of REG_i.

- One or more write invocations in the sequence $\mathsf{w}, \ldots, \mathsf{w}'$ write REG_i. As the value obtained by r_i is the ith field of R, it follows from the regularity of R that it is the value of REG_i before or after any of these write invocations, which proves that r_i satisfies the regularity property. □

The second lemma is on finite sequences of values. It shows that a "shorter" sequence B satisfying some properties can always be extracted from any sequence of values A. B is called a *digest* of A. The properties of B are the following: A and B have the same first and last elements, an element appears at most once in B, and two consecutive elements of B are also consecutive in A. As an example let $A = v_1, v_2, v_1, v_3, v_4, v_2, v_4, v_5$ (where the v_i are values). The sequence $B = v_1, v_3, v_4, v_5$ is a digest of A.

Lemma 26 *Let $A = a_1, a_2, \ldots, a_n$ be a finite sequence of values. For any such sequence there exists a sequence $B = b_1, \ldots, b_m$ of values of A such that*:

- $b_1 = a_1 \wedge b_m = a_n$,
- $(b_x = b_y) \Rightarrow (x = y)$,
- $\forall j : 1 \leq j < m : \exists i : 1 \leq i < n : b_j = a_i \wedge b_{j+1} = a_{i+1}$.

Proof The proof is by trivial induction on n. If $n = 1$, we have $B = a_1$. If $n > 1$, let $B = b_1, \ldots, b_m$ be a digest of $A = a_1, a_2, \ldots, a_n$. A digest of $a_1, a_2, \ldots, a_n, a_{n+1}$ can be constructed as follows:

– If $\forall j \in \{1, \ldots, m\} : b_j \neq a_{n+1}$, then $B = b_1, \ldots, b_m, a_{n+1}$ is a digest of $a_1, a_2, \ldots, a_n$.
– If $\exists j \in \{1, \ldots, m\} : b_j = a_{n+1}$, there is a single j such that $b_j = a_{n+1}$ (this is because any value appears at most once in $B = b_1, \ldots, b_m$). In that case, it is easy to see that $B = b_1, \ldots, b_j$ is a digest of $a_1, \ldots, a_n, a_{n+1}$. □

12.1.2 The Lower Bound Theorem

This theorem, which is due to L. Lamport (1986), asserts that, when we want to construct an SWSR atomic register from bounded regular registers, there is no construction in which the writer only writes and the reader only reads. This means that any such construction must involve two-way communication between the reader and the writer.

Theorem 52 *It is not possible to build an SWSR b-valued atomic register ($b \geq 2$) from a finite number of regular registers that can take a finite number of values and are written only by the writer.*

Proof The proof is by contradiction. Let us assume that it is possible to build an SWSR atomic register R from a finite number of SWSR regular registers, each with a finite value domain. Without loss of generality, let us consider that the atomic register R that is built is binary. Let us first observe that, due to Lemma 25, it is sufficient to consider the case where R is built from a single base regular register (denoted *REG*). The proof considers a possible behavior for the writer and a possible behavior for the reader, and then deduces a contradiction from these behaviors.

A write pattern. Assuming R is initialized to 0, let us consider an execution where infinitely often the writer alternately writes 1 and 0 in R. Let w_i, $i \geq 1$, denote the ith invocation of the operation $R.\mathsf{write}(v)$. This means that $v = 1$ when i is odd and $v = 0$ when i is even.

Invocations of $R.\mathsf{write}(1)$. Each write invocation w_{2i+1} of $R.\mathsf{write}(1)$ is implemented by a sequence of invocations of base write operations on the regular register *REG*. Let $\omega_1, \ldots, \omega_x$ be the sequence of base writes generated by w_{2i+1}, and A_i the corresponding sequence of values defined as follows: its first element a_1 is the value of *REG* before ω_1, its second element a_2 is the value of *REG* just after ω_1 and before ω_2, etc.; its last element a_{x+1} is the value of *REG* after ω_x. Let B_i be a digest derived from A_i (due to Lemma 26 such a digest sequence exists).

As the number of distinct values that *REG* can have is finite, it follows that the number of distinct digest sequences B_i is finite. As the sequence of writes on R is infinite, there is a digest sequence B_i that appears infinitely often. Let $B = b_1, \ldots, b_{y+1}$ ($y \geq 1$) be such a sequence.

Remark Let us observe that there is no constraint on the number of internal states of the writer. This means that all the sequences A_i can be different. It is possible, with an infinite number of internal states, for the writer never to perform the same sequence of base write operations twice. That is why each sequence A_i is replaced by its digest B_i, in such a way that the set of possible digests is finite. This observation is the main motivation for Lemma 26.

Let us observe that, due to the first item of Lemma 26, b_1 is the value of *REG* just before the invocation of an operation $R.\mathsf{write}(1)$. As the writer writes alternately 0 and 1 in R, this means that an invocation of $R.\mathsf{read}()$ all of whose readings of *REG* return the value b_1 has to return 0 as the value of R (i.e., b_1 is one of the low-level encodings that represent the value 0 of R).

The contradiction is provided by showing that b_1 is also a low-level encoding representing the value 1 of R.

Invocations of $R.\mathsf{read}()$. An invocation r of $R.\mathsf{read}()$ is implemented as a sequence of base operations that read *REG*. However, in our quest for a contradiction, we restrict our attention to the scenarios in which each read of *REG* issued by a read invocation r returns the same value. That value is denoted $\lambda(\mathsf{r})$ (notice that $\lambda(\mathsf{r})$ is obtained from a decoding of the value of *REG*). So, we may assume that each invocation r of $R.\mathsf{read}()$ issues a single read on the base regular register *REG*, and that read returns $\lambda(\mathsf{r})$.

Let r and r' be two consecutive read invocations of $R.\mathsf{read}()$. It is possible that these read invocations be such that $\lambda(\mathsf{r}) = b_{j+1}$, $\lambda(\mathsf{r}') = b_j$ (where b_{j+1} and b_j are two consecutive values of the digest B), and both r and r' return 1. An example

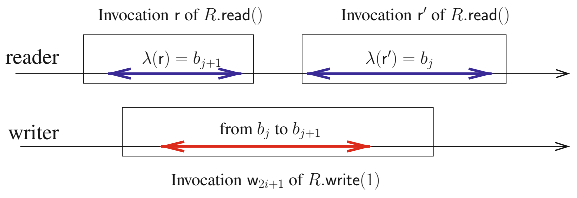

Invocation r of R.read()
Invocation r′ of R.read()
reader
$\lambda(\mathsf{r}) = b_{j+1}$
$\lambda(\mathsf{r}') = b_j$
writer
from b_j to b_{j+1}
Invocation w_{2i+1} of R.write(1)

Fig. 12.1 Two read invocations r and r′ concurrent with an invocation w_{2i+1} of R.write(1)

where this scenario can occur is when both r and r′ are concurrent with an invocation w_{2i+1} of R.write(1) as shown in Fig. 12.1 (where $\lambda(\mathsf{r})$ is used to represent both a read invocation r and the value obtained from *REG* by that read invocation). A write of *REG* that changes its value from a to b is denoted "from a to b"). More explicitly we have the following:

- Let us first look at the constructed register R (i.e., on the side of the semantics provided to the upper layer). As r and r′ are concurrent with a write operation $\mathsf{w}_{2i+1} = R$.write(1), and r returns the new value of R, the immediately following read r′ also has to return the new value, as R is atomic (initial assumption).
- Let us now look at the base register *REG* (i.e., on the side of the implementation). Due to the third item of Lemma 26, when the invocation w_{2i+1} of the R.write(1) operation is executed there are two consecutive base write operations ω_z and ω_{z+1} such that ω_z writes b_j in *REG* and ω_{z+1} writes b_{j+1} in *REG*. As the register *REG* is regular, it is possible that the two consecutive base invocations (that read *REG*) issued by r and r′ while *REG* is concurrently written by the base write invocation ω_{z+1} (that modifies its value from b_j to b_{j+1}) be such that the first obtains the new value b_{j+1} while the second obtains the old value b_j.

The contradiction. Let us now consider a sequence $S = \mathsf{r}_0, \mathsf{r}_1, \ldots, \mathsf{r}_y$ of consecutive invocations of R.read() such that $\lambda(\mathsf{r}_0) = b_{y+1}$, $\lambda(\mathsf{r}_1) = b_y, \ldots, \lambda(\mathsf{r}_y) = b_1$, where $B = b_1, \ldots, b_{y+1}$ is the digest sequence defined above. According to the previous observation applied to the pair of consecutive operations r_0 and r_1, we conclude that both r_0 and r_1 must return the value 1. Applying the same reasoning to the sequence of pairs $(\mathsf{r}_1, \mathsf{r}_2), (\mathsf{r}_2, \mathsf{r}_3), \ldots, (\mathsf{r}_{y-1}, \mathsf{r}_y)$ (notice that this is possible as the length of the sequence B is $y + 1$, see Fig. 12.2), we conclude that r_y has to return the value 1.

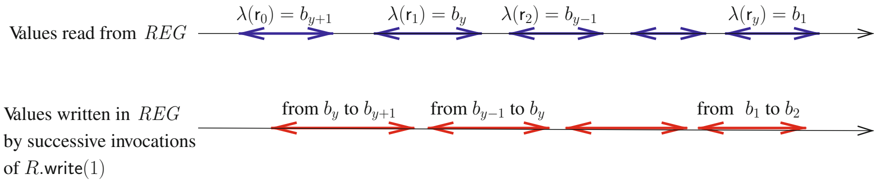

Values read from REG
$\lambda(\mathsf{r}_0) = b_{y+1}$
$\lambda(\mathsf{r}_1) = b_y$
$\lambda(\mathsf{r}_2) = b_{y-1}$
$\lambda(\mathsf{r}_y) = b_1$
Values written in REG by successive invocations of R.write(1)
from b_y to b_{y+1}
from b_{y-1} to b_y
from b_1 to b_2

Fig. 12.2 A possible scenario of read/write invocations at the base level

As $\lambda(\mathsf{r}_y) = b_1$, this means that b_1 is a low-level encoding of the value 1, contradicting the fact it is a low-level encoding of the value 0. □

The previous theorem shows that two-way communication is a necessary requirement to build an SWSR b-bounded atomic register from bounded regular registers. This means that we need at least two control bits: one written by the writer and read by the reader, and one written by the reader and read by the writer. This means that we have the following corollary (the third bit is the bit destined to contain the value of the atomic register):

Corollary 4 *The construction of an SWSR atomic bit from regular bits requires at least three SWSR regular bits.*

As we have seen in the previous chapter, there is a trivial bounded construction of a regular bit from a safe bit. This construction does not require additional registers (it only requires the writer to manage an additional local variable). The combination of this construction with the previous corollary provides the following one.

Corollary 5 *The construction of an SWSR atomic bit from safe bits requires at least three SWSR safe bits* (*two written by the writer and one written by the reader*).

As the construction presented in the next section uses exactly three SWSR regular bits to build an atomic bit, it is optimal in the number of base safe bits.

12.2 A Construction of an Atomic Bit from Three Safe Bits

As just indicated, this section presents an optimal construction of an SWSR atomic bit R from three base SWSR safe bits. The construction, due to J. Tromp (1989), is presented in an incremental way.

The high-level bit R is assumed to be initialized to 0. It is also assumed that each invocation of $R.\mathsf{write}(v)$ changes the value of R. This is done without loss of generality, as the writer of R can locally keep a local copy v' of the last value it has previously written and apply the next $R.\mathsf{write}(v)$ operation only when it modifies the current value of R, i.e., if $v \neq v'$.

12.2.1 Base Architecture of the Construction

From safe to regular registers The three base safe registers are initialized to the initial value of R, i.e., 0. Then, as we will see, the read and write algorithms defining the construction are such that any write applied to a base register X changes its value. So, the sequence of successive values written into any base safe register is the sequence 0, 1, 0, etc. Consequently, to simplify the writing of the construction and to stress the fact it is always updated to the "other" value, the writing of a new value in the base register X is denoted "$X \leftarrow (1 - X)$".

As any two consecutive write operations of a base bit X write different values, it follows that the base register X behaves as a regular register. A read not concurrent with a write returns the last written value, while a read concurrent with a write returns 0 or 1: whatever the value returned, it is the value before the write or a value that is currently being written. It still remains possible for two consecutive reads of X that overlap with the same write that the first obtains the new value of X while the second obtains the previous value (new/old inversion).

Although the writer and the reader of a base register X are different processes, we consider in the text of the read and write algorithms that the writer of X can also read it. This is done without loss of generality as the writer of X can keep a local copy and read it directly.

Internal representation of the constructed atomic register R The three base SWSR safe bits used in the construction of the high-level atomic register R are the following:

- The safe register *REG* is intended to contain the value of the atomic bit that is constructed. It is written by the writer and read by the reader.
- The safe register *WR* is written by the writer to pass control information to the reader.
- The safe register *RR* is written by the reader to pass control information to the writer.

As we can see, *REG* is the data part of the construction, while the pair of safe bits $\langle WR, RR \rangle$ constitutes the data of its control part.

12.2.2 Underlying Principle and Signaling Scheme

As previously suggested, the basic idea on which the construction relies consists in using the pair of control bits (WR, RR) to pass information from the reader to the writer and vice versa in order to prevent new/old inversions. To implement this passing of control information, we choose to have the following scheme:

- To indicate that a new value was written, the writer makes *WR* different from *RR*.
- To indicate that a new value was read, the reader makes *RR* equal to *WR*.

(Let us notice that this choice is arbitrary in the sense that we could have chosen the writer to make the registers equal and the reader to make them different.)

```
operation $R$.write() is    %The value of $R$ changes %
(1)  $REG \leftarrow (1 - REG)$;
(2)  if $WR = RR$ then $WR \leftarrow (1 - WR)$ end if;
(3)  return($ok$)
end operation.

operation $R$.read() is
(4)  if $WR = RR$ then return($val$) end if;
(5)  $aux \leftarrow REG$;
(6)  if $WR \neq RR$ then $RR \leftarrow (1 - RR)$ end if;
(7)  $val \leftarrow REG$;
(8)  if $WR = RR$ then return($val$) end if;
(9)  $val \leftarrow REG$;
(10) return($aux$)
end operation.
```

Fig. 12.3 Tromp's construction of an atomic bit

12.2.3 *The Algorithm Implementing the Operation* R.write()

This algorithm is described in Fig. 12.3. The writer first updates the base register *REG* to its new value (line 1). Then, it strives to establish $WR \neq RR$ to inform the reader that a new value was written (line 2).

As we are about to see, the reader strives to establish $RR = WR$ before reading *REG*. This is how it informs the writer that it is about to read the regular bit *REG* implementing *R*. This allows the writer to know that its previous value was read. Interestingly, the reader itself uses the predicate $RR = WR$ to check if the value it has read from *REG* can be returned as the value of *R*.

12.2.4 *The Algorithm Implementing the Operation* R.read()

The read algorithm is much more involved than the write algorithm. It is described in Fig. 12.3. To make it easier to understand, this section adopts an informal and incremental presentation.

The construction: step 1 As we have seen previously, before reading the base register *REG*, the reader modifies *RR* (if needed) in order to obtain $wr = RR$ where *wr* represents the value it obtains from the regular register *WR*. Then, after it has read the base register *REG* (let val be the value it has obtained), the reader reads again *WR* (that meanwhile may have been modified by the writer) to check the predicate $WR = RR$. If it is true, it returns the value val, as from its point of view, *WR* has not changed between the two consecutive reads of it, and consequently *REG* should not have been changed. This is described by the following sequence of statements (the line numbers refer to the final version of the read algorithm).

```
(6) if $WR \neq RR$ then $RR \leftarrow (1 - RR)$ end if; % Strive to establish $WR = RR$ %
(7) $val \leftarrow REG$;
(8) if $WR = RR$ then return($val$) end if.
```

The construction: step 2 This sketch of a read algorithm is incomplete as it does not return a value when the reader does not find $WR = RR$ in the last statement. A simple way to solve this problem consists in adding at the end a return() statement with an appropriate value. Returning val would make the flags WR and RR useless (and we know that such flags are required by Theorem 52). The idea is then to choose a conservative value, namely the value of REG at the very beginning of the read operation. The previous sequence of statements is consequently enriched as follows with lines 5 and 10 , where aux is a local variable whose scope is limited to the current invocation of R.read().

```
(5)  $aux \leftarrow REG$; % Conservative value %
(6)  if $WR \neq RR$ then $RR \leftarrow (1 - RR)$ end if; % Strive to establish $WR = RR$ %
(7)  $val \leftarrow REG$;
(8)  if $WR = RR$ then return($val$) end if;
(10) return($aux$).
```

It is easy to verify that the register R constructed by the read operation just described (lines 5–8 and line 10) and the write operation (lines 1–3) is regular. A read of R that is not concurrent with a write of R returns the current value of R, and a read of R that is concurrent with one or more writes of R returns 0 or 1. Unfortunately, this construction does not build an atomic register R as it is still possible to have new/old inversions.

The construction: final step A way to prevent new/old inversions is for the reader to help itself by requiring some invocations of R.read() to help future invocations of R.read() by saving in its local memory (namely in its local variable val, whose scope is now the whole execution) a value previously obtained from REG. That value can be returned in appropriate situations to prevent new/old inversions as far as R is concerned. The final read algorithm is described in Fig. 12.3. More precisely, we have the following:

- When it invokes R.read(), the reader first checks the predicate $WR = RR$ (line 4). If it is true, the reader considers that no invocation of R.write() was issued since its last reading of REG. In that case, the reader returns the last value it has previously obtained from REG, namely the value that was saved in the persistent local variable val.

  Let us observe that this allows some read operations to return without accessing the data register REG.

- Before executing return(aux) (line 10) the reader reads REG and saves its value in val (new line 9). This allows val to contain a "fresh" value of REG (as the content of REG may have been changed since its previous reading).

Remark The reader should be conscious of the fact that the previous presentation relies mainly on intuitive explanations. As the base registers are only safe bits (that behave as regular bits), this kind of intuition could easily be flawed. That is why the only way to be convinced that the construction is correct consists in providing a proof.

12.2.5 Cost of the Construction

As far as memory space is concerned, the cost of the construction is three SWSR safe bits plus a permanent local variable (val).

As in previous chapters, the time complexity of the R.read() and R.write() operations is measured by the the maximal and minimal numbers of accesses to the base registers. Let us recall that the writer (reader) does not have to read *WR* (*RR*), as it can keep a local copy of that register.

- R.write(v): maximal cost: 3; minimal cost: 2.
- R.read(): maximal cost: 7; minimal cost: 1.

The minimal cost is realized when the same type of operation (i.e., read or write) is repeatedly executed while the operation of the other type is not invoked.

Let us remark that we have assumed that, if R.write(v) and R.write(v') are two consecutive invocations of the write operation, we have $v \neq v'$. This means that, if the upper layer issues two consecutive write invocations with $v = v'$, the cost of the second one is 0, as it is skipped and consequently there is no access to a base register.

12.3 Proof of the Construction of an Atomic Bit

12.3.1 A Preliminary Theorem

Notations The following theorem considers an SWMR multi-valued register. It will be used in the following in the particular case of an SWSR binary register. Let r be an invocation of R.read(), and v the value read by r. As in previous chapters, $\pi(\mathsf{r})$ denotes the write invocation that writes v into R. The other definitions and notations used in the following have been introduced in Chap. 4.

Theorem 53 *Let $\widehat{H}$ be an execution history of an SWMR register R. $\widehat{H}$ is atomic (or linearizable) if and only if it satisfies the three following properties (where* r, r_1*, and* r_2 *are invocations of* R.read() *and* w *is an invocation of* R.write()):

- $A0$: $\forall \mathsf{r}$: $\neg(\mathsf{r} \rightarrow_H \pi(\mathsf{r}))$ ($\widehat{H}$ is an execution history).
- $A1$: $\forall \mathsf{r}, \mathsf{w}$: $(\mathsf{w} \rightarrow_H r) \Rightarrow \big(\pi(\mathsf{r}) = \mathsf{w} \vee \mathsf{w} \rightarrow_H \pi(\mathsf{r})\big)$ (no read obtains an overwritten value).

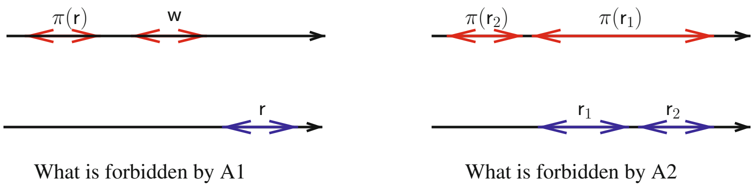


Fig. 12.4 What is forbidden by the properties A1 and A2

- *A*2: $\forall \mathsf{r}_1, \mathsf{r}_2$: $(\mathsf{r}_1 \rightarrow_H \mathsf{r}_2) \Rightarrow \big(\pi(\mathsf{r}_1) = \pi(\mathsf{r}_2) \vee \pi(\mathsf{r}_1) \rightarrow_H \pi(\mathsf{r}_2)\big)$ (no new/old inversion).

This theorem states that an execution of an SWMR register is atomic if and only if it is a feasible history (A0), no read invocation obtains an overwritten value (A1), and there is no new/old inversion (A2). The forbidden situations are described in Fig. 12.4 (r_1 and r_2 can be issued by the same process or different processes; the important point is that r_2 starts after r_1 has terminated).

Proof The "only if" part follows directly from the fact that, if one of the assertions A0, A1, or A2 is not satisfied, it is clearly impossible to construct a legal sequential history $\widehat{S}$ satisfying $\rightarrow_H \subseteq \rightarrow_S$.

Let $\widehat{H}$ be an execution history at the level of the invocations of the operations R.read() and R.write(). The proof of the "if" part consists in finding a total order $\widehat{S}$ on all operations in $\widehat{H}$ that is legal and respects the partial order $\rightarrow_H$ on the operation invocations defined by $\widehat{H}$.

A sequential history $\widehat{S}$ is built as follows. We start from the invocations of the operation R.write() because, as there is a single writer, they provide a germ to build a total order. We then add the invocations to the operation R.read() into this total order in such a way that each write invocation w is immediately followed by the read invocations r such that $\pi(\mathsf{r}) = \mathsf{w}$. Moreover, if $\pi(\mathsf{r}_1) = \pi(\mathsf{r}_2) = w$, we place first the read invocation that started first (i.e., the one whose start event precedes the start event of the other; let us remind that, at the event level, an execution history is defined by a total order). This total order on the operation invocations provides a sequential history $\widehat{S}$ that (1) is legal (a read obtains the value of the last preceding write invocation) and (2) is equivalent to $\widehat{H}$ (as it is made up of the same operations as $\widehat{H}$).

It remains to show that $\rightarrow_H \subseteq \rightarrow_S$. As $\widehat{H}$ contains all completed invocations of $\widehat{H}$, $\rightarrow_H \subseteq \rightarrow_S$ will imply that $\rightarrow_H$ is equivalent to $\rightarrow_S$. We consider the four possible cases ($w1$ and $w2$, and $r1$ and $r2$ denote write and read perations, respectively).

1. $\mathsf{w}_1 \rightarrow_H \mathsf{w}_2$. Due to the very construction of $\widehat{S}$ (that considers the order on the write invocations as imposed by the writer), we have $\mathsf{w}_1 \rightarrow_S \mathsf{w}_2$.
2. $\mathsf{r}_1 \rightarrow_H \mathsf{r}_2$. Due to A2, we have $\pi(\mathsf{r}_1) = \pi(\mathsf{r}_2)$ or $\pi(\mathsf{r}_1) \rightarrow_H \pi(\mathsf{r}_2)$.

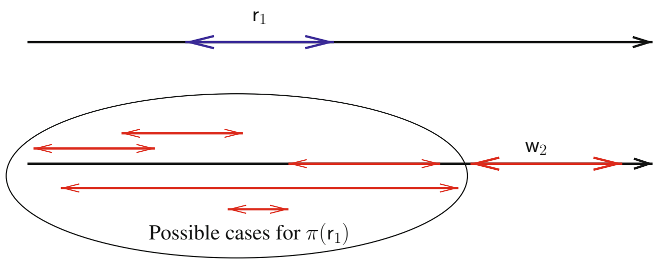


Fig. 12.5 $\pi(r_1) \rightarrow_H w_2$

- Case $\pi(r_1) = \pi(r_2)$. As r_1 started before r_2 (case assumption), due to the way $\widehat{S}$ is constructed, r_1 is ordered before r_2 in $\widehat{S}$, and we have consequently $r1 \rightarrow_S r2$.
- Case $\pi(r_1) \rightarrow_H \pi(r_2)$. As (a) ($r_1$ and r_2 are placed just after $\pi(r_1)$ and $\pi(r_2)$, respectively, and (b) $\pi(r_1)1 \rightarrow_S \pi(r_2)$ (see the first item), the construction of $\widehat{S}$ ensures $r_1 \rightarrow_S r_2$.

3. $r_1 \rightarrow_H w_2$. Due to A0, either $\pi(r_1)$ is concurrent with r_1 or $\pi(r_1) \rightarrow_H r_1$. This means that $\pi(r_1)$ starts (event $inv[\pi(r_1)]$) before r_1 terminates (event $resp[r_1]$ in the total order defined by $\widehat{H}$ on all the events) (i.e., $inv[\pi(r_1)] <_H resp[r_1]$). Moreover, as $r_1 \rightarrow_H w_2$, we conclude that the event $resp[r_1]$ precedes the event $inv[w_2]$ ($resp[r_1] <_H inv[w_2]$). It then follows that, when we consider the total order on the events, we have that $inv[\pi(r_1)]$ precedes $inv[w_2]$ ($inv[\pi(r_1)] <_H inv[w_2]$). As the write invocations are totally ordered, this implies $\pi(r_1) \rightarrow_H w_2$ (see Fig. 12.5). We can then conclude $\pi(r_1) \rightarrow_S w_2$ from the first case.
   As r_1 is ordered in $\widehat{S}$ just after $\pi(r_1)$ (i.e., before the following write invocation) and $\pi(r_1) \rightarrow_S w_2$, we have $r_1 \rightarrow_S w_2$.
4. $w_1 \rightarrow_H r_2$. Due to A1 we have $\pi(r_2) = w_1$ or $w_1 \rightarrow_H \pi(r_2)$.
   - Case $\pi(r_2) = w_1$. As r_2 is placed just after $\pi(r_2)$ in $\widehat{S}$, we have $\pi(r_2) = w1 \rightarrow_S r_2$.
   - Case $w1 \rightarrow_H \pi(r2)$. As $w_1 \rightarrow_H \pi(r_2) \Rightarrow w_1 \rightarrow_S \pi(r_2)$ (first item), and $\pi(r_2) \rightarrow_S r_2$ (r_2 is ordered just after $\pi(r_2)$ in $\widehat{S}$), we obtain (by transitivity of $\rightarrow_S$) $w_1 \rightarrow_S r_2$. □

12.3.2 Proof of the Construction

Theorem 54 *Let $\widehat{H}$ be any execution history of an SWSR register R built from three safe SWSR bits by Tromp's construction Fig. 12.3. $\widehat{H}$ is linearizable.*

Proof Let $\widehat{H}$ be an execution history. Let us recall that $<_H$ denotes the total order on its start and response events, while $\rightarrow_H$ denotes the relation induced by $<_H$ on its invocations of the operations R.write() and R.read() (see Chap. 4). To show that $\widehat{H}$ is atomic (linearizable) we show that it satisfies the assertions A0, A1, and A2 defined in the statement of Theorem 53. Then, the result follows as a consequence of that theorem.

To distinguish the invocations of the operations R.read() and R.write() (that, as previously, we denote r and w) from the read and write invocations on the base registers (e.g.," $RR \leftarrow (1 - RR)$", "$aux \leftarrow REG$"), the latter are called *actions*. The history defined from the action start and response events is denoted $\widehat{L}$ ($<_L$ denotes the total order on its events and $\rightarrow_L$ the corresponding relation induced on its read/write invocations; without loss of generality, $<_L$ is assumed to contain all the start and response events defining $\widehat{H}$).

Moreover, r being an invocation of R.read() and loc the local variable (aux or val) containing the value returned by r (at line 4, 8 or 10), ρ_{r} denotes the last read action "$loc \leftarrow REG$" executed before r returns. More explicitly, we have the following:

- If r returns at line 10, ρ_{r} is the read action "$aux \leftarrow REG$" executed at line 5 of r,
- If r returns at line 8, ρ_{r} is the read action "$val \leftarrow REG$" executed at line 7 of r, and finally
- If r returns at line 4, ρ_{r} is the read action "$val \leftarrow REG$" executed at line 7 or 9 by a previous invocation of R.read().

As each base register (REG, RR, and WR) behaves as a regular register, each read action ρ_{r} has a corresponding write action, denoted $\pi(\rho_{\mathsf{r}})$. Finally, given a read invocation r and its associated read action ρ_{r}, $\pi(\mathsf{r})$ denotes the invocation of R.write() which includes the write action $\pi(\rho_{\mathsf{r}})$. This means that the value returned by the read invocation r was written in the base register REG by the action "$REG \leftarrow 1 - REG$" issued by the invocation of R.write() denoted $\pi(\mathsf{r})$. For notational convenience we say $a \in A$ when a is an action of the operation A.

Proof of A0 ($\widehat{H}$ is an execution history). Let us first observe the following:

- As REG behaves as a regular register, a value that is read is a value that was previously written or is concurrently written. It follows that the read action ρ_{r} cannot precede the corresponding write action $\pi(\rho_{\mathsf{r}})$, which means that we have $inv[\pi(\rho_{\mathsf{r}})] <_L resp[\rho_{\mathsf{r}}]$.
- Due to the very definition of r and ρ_{r}, we have $resp[\rho_{\mathsf{r}}] <_L resp[r]$.

It follows that $inv[\pi(\rho_{\mathsf{r}})] <_L resp[r]$, from which we conclude $\neg(resp[r] <_L inv[\pi(\rho_{\mathsf{r}})])$.

Let us now reason by contradiction. Let us assume that A0 is violated; i.e., at the level defined by the invocations r and $\pi(\mathsf{r})$, we have $\mathsf{r} \rightarrow_H \pi(\mathsf{r})$. This translates at the action event level as $resp[\mathsf{r}] <_L inv[\pi(\rho_{\mathsf{r}})]$, which contradicts the property that was previously established and completes the proof of A0.

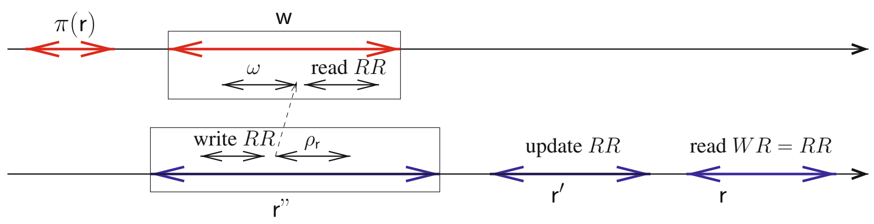


Fig. 12.6 ρ_r belongs neither to r nor to r′

Proof of A1 (no invocation of $R.\mathsf{read}()$ returns an overwritten value).
Let us first observe that, as there is a single writer at the level of the $R.\mathsf{read}()$ and $R.\mathsf{write}(v)$ operations, no two write invocations can be concurrent; i.e., $\mathsf{w} \rightarrow_H \pi(\mathsf{r})$ is equivalent to $\neg(\pi(\mathsf{r}) \rightarrow_H \mathsf{w})$. The proof of A1 is by contradiction. Let us assume that there is a write invocation $\mathsf{w} \neq \pi(\mathsf{r})$ such that $\pi(\mathsf{r}) \rightarrow_H \mathsf{w} \rightarrow_H \mathsf{r}$. If there are several such write operations, let w be the last one before r, i.e., $\nexists\, \mathsf{w}'$: $\mathsf{w} \rightarrow_H \mathsf{w}' \rightarrow_H \mathsf{r}$. The proof is based on two claims.

Claim C1. ρ_r cannot be a read action of the read invocation r (i.e., $\rho_\mathsf{r} \notin \mathsf{r}$).
Proof of the claim. Let us recall that $\pi(\rho_\mathsf{r}) \in \pi(\mathsf{r})$ (by definition) (see Fig. 12.6). Let ω be the action “$REG \leftarrow (1 - REG)$” issued by the invocation w ($\omega \in \mathsf{w}$). Combined with the case assumption $\pi(\mathsf{r}) \rightarrow_H \mathsf{w}$, we obtain $\pi(\rho_\mathsf{r}) \rightarrow_L \omega$. As (by definition) ρ_r obtains the value written by $\pi(\rho_\mathsf{r})$, we have $\neg(\omega \rightarrow_L \rho_\mathsf{r})$ (otherwise, as the base register *REG* behaves as a regular register, ρ_r could not obtain the value written by $\pi(\rho_\mathsf{r})$). It then follows that $inv[\rho_\mathsf{r}] <_L resp[\omega]$. As $\omega \in \mathsf{w}$ and $w \rightarrow_H \mathsf{r}$, we have $inv[\rho_\mathsf{r}] <_L resp[\mathsf{w}] <_L inv[\mathsf{r}]$. As ρ_r started before r and both are executed by the same process, we have $\rho_\mathsf{r} \notin \mathsf{r}$. *End of the proof of the claim.*

It follows from that claim ($\rho_\mathsf{r} \notin \mathsf{r}$) and the definition of ρ_r that the read invocation r returns a value at line 4, which means that it has seen $WR = RR$. However, after it has executed ω, the write invocation has read *RR* in order to set *WR* different from *RR* if they were seen equal. As $\mathsf{w} \rightarrow_H \mathsf{r}$ and $\nexists\, \mathsf{w}'$: $\mathsf{w} \rightarrow_H \mathsf{w}' \rightarrow_H \mathsf{r}$ (assumption), it follows that *RR* was modified before the read invocation r starts. Moreover, *RR* can only be modified by a read invocation at line 6. Let r′ be that read invocation; as there is a single process invoking the operation $R.\mathsf{read}()$, we have $\mathsf{r}' \rightarrow_H \mathsf{r}$.

Claim C2. $\rho_\mathsf{r} \notin \mathsf{r}'$.
Proof of the claim. Let r″ be the read invocation that contains ρ_r. We show $\mathsf{r}'' \neq \mathsf{r}'$. Let us observe the following facts (Fig. 12.6):

- If r'' updates *RR*, it does so at line 6 (i.e., before executing ρ_r at line 8).
- $inv[\rho_\mathsf{r}] <_L resp[\omega]$ (this has been shown above; it is indicated by a dotted arrow in Fig. 12.6).
- w reads RR after having executed ω (code of the write algorithm).

It follows from these observations that, if r″ writes into RR, it does so before w reads RR. Hence, r″ cannot change the value of RR (to establish $RR = WR$) after w has

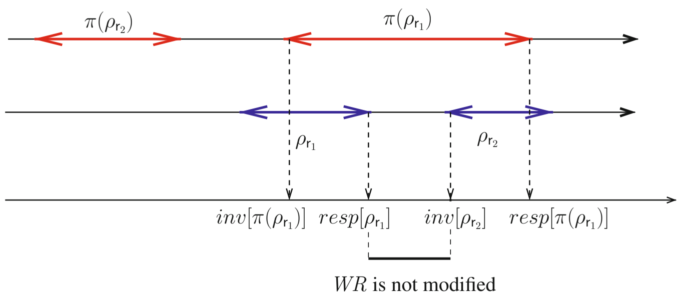


Fig. 12.7 A new/old inversion on the regular register *REG*

read RR or while it is reading it (to establish $RR \neq WR$). Consequently r'' and r' (which by definition updates *RR*) are different and we have $\rho_{\mathsf{r}} \notin \mathsf{r}'$. *End of the proof of the claim.*

As it modifies RR when it executes r', the reader process executes also line 7 of r' ($val \leftarrow REG$) before executing r (this follows from the code of the read algorithm). But, as $\rho_{\mathsf{r}} \notin \mathsf{r}'$, this read of *REG* action contradicts the definition of ρ_{r} (which states that ρ_{r} is the last reading action "$val \leftarrow REG$" executed before the invocation r starts). This contradiction completes the proof of assertion A1.

Proof of A2 (no new/old inversion).
The proof is again by contradiction. Let r_1 and r_2 be two invocations of $R.\mathsf{read}()$ such that $\mathsf{r}_1 \rightarrow_H \mathsf{r}_2$ and $\pi(\mathsf{r}_2) \rightarrow_H \pi(\mathsf{r}_1)$ (see the right side of Fig. 12.4). Without loss of generality, let us assume that, for a given r_2, r_1 is the first such read invocation. This means that (a) if r_1 returns at line 8 or 10, ρ_{r1} is a read action belonging to r_1 and (b) if r_1 returns at line 4,then ρ_{r1} is a read action in the immediately preceding read invocation. Moreover, as $\pi(\mathsf{r}_2) \neq \pi(\mathsf{r}_1)$, we have $\rho_{r1} \neq \rho_{r2}$. So, we have either $\rho_{\mathsf{r}_1} \rightarrow_L \rho_{\mathsf{r}_2}$ or $\rho_{\mathsf{r}_2} \rightarrow_L \rho_{\mathsf{r}_1}$.

- Case $\rho_{\mathsf{r}_2} \rightarrow_L \rho_{\mathsf{r}_1}$.
  As ρ_{r_1} precedes or belongs to r_1 and $\mathsf{r}_1 \rightarrow_H \mathsf{r}_2$, we have $resp[\rho_{\mathsf{r}_1}] <_L inv[\mathsf{r}_2]$. Combining this with the case assumption we obtain $\rho_{\mathsf{r}_2} \rightarrow_L \rho_{\mathsf{r}_1} \rightarrow_L \mathsf{r}_2$, which contradicts the fact that ρ_{r_2} is the last action "$val \leftarrow REG$" executed before r_2 starts. So, the case $\rho_{\mathsf{r}_2} \rightarrow_L \rho_{\mathsf{r}_1}$ is not possible.

- Case $\rho_{\mathsf{r}_1} \rightarrow_L \rho_{\mathsf{r}_2}$.
  By definition $\pi(\rho_{\mathsf{r}_1}) \in \pi(\mathsf{r}_1)$ and $\pi(\rho_{\mathsf{r}_2}) \in \pi(\mathsf{r}_2)$. As $\pi(\mathsf{r}_2) \rightarrow_H \pi(\mathsf{r}_1)$, we have $\pi(\rho_{\mathsf{r}_2}) \rightarrow_L \pi(\rho_{\mathsf{r}_1})$. However, $\pi(\rho_{\mathsf{r}_2}) \rightarrow_L \pi(\rho_{\mathsf{r}_1})$ and $\rho_{\mathsf{r}_1} \rightarrow_L \rho_{\mathsf{r}_2}$ (Fig. 12.7) characterize a new/old inversion with respect to the base register *REG* (whose behavior is regular). For this new/old inversion to occur, we need to have $inv[\pi(\rho_{\mathsf{r}_1})] <_L resp[\rho_{\mathsf{r}_1}]$ and $inv[\rho_{\mathsf{r}_2}] <_L resp[\pi(\rho_{\mathsf{r}_1})]$ (because both ρ_{r_1} and ρ_{r_2} have to overlap with $\pi(\rho_{\mathsf{r}_1})$ in order to have a new/old inversion, Fig. 12.7). As $\pi(\rho_{\mathsf{r}_1})$ is a base action that updates *REG*, and as it is the same process that updates *REG* and *WR*, this means that the value of the base register *WR* does not change while it

is updating *REG*, from which we conclude that *WR* does not change between the events $resp[\rho_{\mathsf{r}_1}]$ and $inv[\rho_{\mathsf{r}_2}]$. Let P denote that property. We consider three cases according to the line at which r_1 returns:

– r_1 returns at line 10.
  Then, ρ_{r1} is "$aux \leftarrow REG$" at line 2 of r_1. We have the following:

  ∗ Since $\rho_{\mathsf{r}_1} \rightarrow_L \rho_{\mathsf{r}_2}$ and r_1 returns at line 10, ρ_{r_2} can only be the read at line 9 of r_1 or a later read action.

  ∗ Since r_1 executes all the actions of the read invocation, at line 6 it makes *RR* equal to *WR* if they were not so already. As r_1 returns at line 10, it necessarily sees *RR* different from *WR* at line 8 (otherwise, it would have returned at that line).

  It follows from these two observations that *WR* was modified between line 5 (execution of ρ_{r_1}) and line 9 of r_1 (which is ρ_{r_2} or precedes it). This contradicts property P, which terminates the proof for this case.

– r_1 returns at line 8.
  Then, ρ_{r_1} is the action "$val \leftarrow REG$" executed by r_1 at line 7 and r_1 sees $RR = WR$ at line 8. Since $\rho_{\mathsf{r}_1} \rightarrow_L \rho_{\mathsf{r}_2}$, r_2 does not return at line 4 (for r_2 to return at line 4, we need to have $RR = WR$ at line 4 of r_2, which means that we would then have $\rho_{\mathsf{r}_1} = \rho_{\mathsf{r}_2}$). Consequently, r_2 sees $RR \neq WR$ when it executes line 4, andρ_{r_2} is line 5 or line 7 of r_2. It follows that *WR* was modified between ρ_{r_1} and ρ_{r_2}: a contradiction with property P, which proves the case.

– r_1 returns at line 4.
  In this case, ρ_{r_1} is line 7 or line 9 of the read invocation that precedes r_1. The reasoning is the same as in the previous case. Since $\rho_{\mathsf{r}_1} \rightarrow_L \rho_{\mathsf{r}_2}$, r_2 does not return at line 4, from which we conclude that it sees $RR \neq WR$ when it executed line 4. It follows that *WR* was modified between ρ_{r_1} and ρ_{r_2}, which contradicts property P and proves the case. □

12.4 Summary

This chapter has presented two fundamental results. The first is a theorem due to Lamport stating that any bounded construction of an SWSR atomic bit from safe bits requires that both the reader and the writer write into the shared memory. This means that the reader has to send information to the writer to cope with the fact that the shared memory is bounded. The second result is an optimal bounded construction of an SWSR atomic bit from three safe bits. Combined with constructions presented in the previous chapter, this allows for bounded constructions of SWSR b-valued atomic registers and unbounded constructions of MWMR unbounded atomic registers.

12.5 Bibliographic Notes

- The main theorem stating that a bounded construction of an SWSR atomic bit from a bounded number of safe bits requires that the reader writes the shared memory is due to L. Lamport [189]. The proof given here is inspired from [70].
- The first bounded construction proposed to build an atomic bit from safe bits was due to L. Lamport in 1986. This construction is described in his seminal paper [190].
- As indicated, the optimal construction of an atomic bit from three safe bits presented here is due to J. Tromp [265].
- A short survey on the construction of atomic registers can be found in [177].

12.6 Exercise

1. Considering the write algorithm described in Fig. 12.3 and the read algorithm described in step 2 of the construction of an atomic bit (Sect. 12.2.4), show that the register R that is built is not atomic. To that end, find an execution where there are new/old inversions.

Chapter 13
Bounded Constructions of Atomic b-Valued Registers

The previous chapter has shown how to construct an SWSR atomic bit from a bounded number (three) of safe bits. Moreover, a bounded construction, due to K. Vidyasankar, of a b-valued atomic register (i.e., a register that can take b different values) from atomic bits was presented in Chap. 11. It is consequently possible to obtain an SWSR b-valued atomic register from a bounded number of SWSR safe bits. However, stacking these two constructions requires $O(b)$ safe bits, i.e., a number of safe bits linear with respect to the size of the value domain of the atomic register we want to construct.

This chapter presents two efficient bounded constructions of an SWSR b-valued atomic register from a constant number of atomic bits and a constant number of b-valued safe registers each made up of $\lceil \log_2 b \rceil$ safe bits. As an atomic bit can be built from three safe bits, these constructions (due to J. Tromp and K. Vidyasankar) require only $O(\lceil \log_2 b \rceil)$ safe bits and are consequently efficient.

As in the previous chapters, the read and write operations associated with the constructed SWSR atomic b-valued register R are denoted R.read() and R.write(), respectively.

Keywords Atomic bit · Atomic register construction · Collision-freedom · Finite state automaton · Linearizability · Pure/impure buffer · Safe buffer · Switch · SWSR register · Wait-freedom

13.1 Introduction

Structure of the constructions The internal representation of each construction is made up of a control part and a data part.

- The data part consists of a set of registers called *buffers*, each composed of $\lceil \log_2 b \rceil$ safe bits. These buffers are managed in such a way that, at any time, one of them contains the last value that was written into the high-level register R (the other ones containing older values or the value that is currently written by the writer). The bits of a buffer can be read and written in any order, and there is no guarantee on the value obtained from a buffer when it is concurrently read and written.

- The control part consists of atomic bits which implement *switches*. These switches are used to direct the writer to write in a given buffer and the reader to read from a buffer which contains the last value that was written into R.

Pure versus impure buffers The two constructions presented in this chapter differ in the way they use buffers. The first uses *pure* buffers, while the second uses *impure* buffers. A buffer is pure if it is never accessed concurrently by the reader and the writer. Otherwise, it is impure. A pure buffer is also called a *collision-free* buffer.

Space and time complexities of the constructions The two constructions exhibit an interesting tradeoff between the number of buffers and the cost of a read (or write) operation.

- The first construction is due to J. Tromp (1989). Its switching mechanism ensures collision-freedom on the operation invocations issued on a same buffer. To that end it needs four buffers (i.e., $4\lceil\log_2 b\rceil$ safe bits). As there is no read/write collision, an invocation of $R.\mathsf{read}()$ or $R.\mathsf{write}()$ needs to access a single buffer only once. The switch mechanism ensuring collision-freedom is implemented with four atomic bits. Hence, its control part needs $4 \times 3 = 12$ safe bits.

  It follows that Tromp's construction requires $4\lceil\log_2 b\rceil + O(1)$ safe bits and a that read or write operation involves $\lceil\log_2 b\rceil + O(1)$ bit accesses ($\log_2 b$ accesses to the safe bits of the single buffer that is accessed plus $O(1)$ accesses to the safe bits implementing the switch).

- The second construction is due to K. Vidyasankar (1990). It uses only three buffers, each made up of $\lceil\log_2 b\rceil$ safe bits. This construction does not prevent a read and a write operation on the same buffer from colliding. To ensure that a read operation returns a correct value, it requires the read and the write operations to read or write one or two buffers according to the value of the switch which is made up of three atomic bits (i.e., $3 \times 3 = 9$ safe bits).

  So, the space complexity of Vidyasankar's construction is $3\lceil\log_2 b\rceil + 9$ safe bits, and the time complexity of a read or write operation can be up to $2\lceil\log_2 b\rceil + O(1)$ ($2\lceil\log_2 b\rceil$ accesses to safe bits of the two buffers that are accessed plus $O(1)$ accesses for the safe bits of the switch).

Remark From a practical point of view, the techniques developed in this chapter can be used to provide wait-free implementations of read/write objects such as a shared file or a read/write web page. The shared memory or the disks supporting the buffers implementing the file or the web page can be accessed without requiring particular constraints (such as a predefined access order) on the way the blocks of the disks or the words of the shared memory have to be accessed.

13.2 A Collision-Free (Pure Buffers) Construction

As already indicated, this first construction is due to J. Tromp (1989).

13.2.1 Internal Representation of the Atomic b-Valued Register R

As just indicated the internal representation of the register R consists of four buffers plus switches built from four atomic bits. The global structure is represented in Fig. 13.1.

The four buffers The four buffers are kept as an array denoted $BUF[0..1, 0..1]$. Each buffer $BUF[i, j]$ is initialized to the initial value of the constructed register R. The buffers are divided into two groups, denoted 0 and 1. The group $i \in \{0, 1\}$ is composed of the buffers $BUF[i, 0]$ and $BUF[i, 1]$.

The switch The switch is actually a two-level switch composed of four atomic bits denoted WR, RR, $D[0]$ and $D[1]$ (each initialized to 0). WR, $D[0]$, and $D[1]$ are written by the writer to convey control information to the reader, while RR is written by the reader to pass control information to the writer.

- The atomic bit RR denotes the buffer group from which the reader has to read.
- Similarly, the atomic bit WR denotes the last group in which a value was written.
- The atomic bits $D[0]$ and $D[1]$ indicate which is the last buffer written in the corresponding group.

13.2.2 Underlying Principle: Two-Level Switch to Ensure Collision-Free Accesses to Buffers

- To avoid collision in a buffer, the writer uses the switch (as defined by the values of WR and RR). It first makes WR different from RR in order to have a chance not to write in a buffer of the same group where it sees the reader. Then it writes

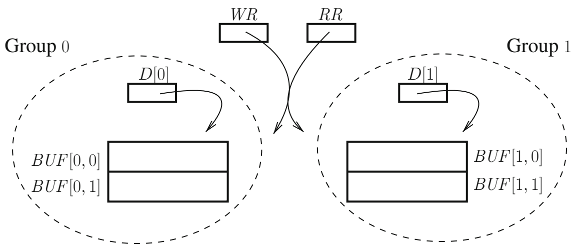


Fig. 13.1 Buffers and switch in Tromp's construction

the buffer $BUF[WR, D[WR]]$, and consequently $D[WR]$ denotes the most recently written buffer in the group WR.

- The reader on its side first makes RR equal to WR in order to read from the last written group. It then reads the current value of the buffer $BUF[RR, D[RR]]$.

Due to asynchrony, it is nevertheless possible that, despite the first-level switch implemented by the atomic bits RR and WR, the reader and the writer access concurrently the same group of buffers. The atomic bits $D[0]$ and $D[1]$ are then used to implement a second-level switch which prevents the reader and the writer from accessing simultaneously the same buffer of a group.

Hence, the core of the construction lies in the design of a two-level switch mechanism ensuring collision-freedom. As just suggested, the four atomic bits RR, WR, $D[0]$, and $D[1]$ implement this switch. RR and WR are used to direct a process to a given group i (0 or 1) of buffers, while the atomic bit $D[i]$ is used inside the group i to prevent collision in the same buffer in the case where both processes would have been directed to the same group.

13.2.3 The Algorithms Implementing the Operations R.write() *and* R.read()

Local variables The writer process manages three local variables: wr, which is a local copy of WR, and $d[0]$ and $d[1]$, where $d[wr]$ ($wr \in \{0, 1\}$) is a local copy of the direction $D[wr]$. On the reader side, rr is a local copy of RR. These local copies are initialized to the same value as their original counterpart, and managed as described in the read and write algorithms.

To get a better intuition of the construction, the reader (1) can first consider the case where the read and the write algorithms do not execute concurrently and (2) always reason as if, at any time, there is at most one access to an atomic bit (this follows from the fact that accesses to atomic bits are linearizable).

The algorithm implementing the operation R.write() The algorithm of the writer is described in lines 1–9 of Fig. 13.2. At the end of a write, the writer manages to have $WR \neq RR$, which is how it informs the reader that a new value was written. The aim of this control information, passed through the pair of atomic bits $\langle WR, RR\rangle$, is to allow the next read invocation to obtain the last written value by reading $BUF[WR, D[WR]]$.

The writer starts by reading RR (the group from which the reader is assumed to read) and compares with to WR (line 1), which is then equal to wr (this equality follows from the fact that only the writer writes WR, line 4). There are two cases according to the value of the first-level switch made up of the atomic bits RR and WR.

- $RR = WR$.
  As (if it is reading) the reader is reading a buffer in the group RR, the first-level switch indicates in that case that the reader is not accessing the other group (the

```
operation R.write(v) is                                   % state $w_0$ %
(1)  if $(RR = wr)$                                        % transition to $w_1$ or $w_2$ %
(2)      then $wr \leftarrow (1 - wr)$;                    % state $w_1$ %
(3)           $BUF[wr, d[wr]] \leftarrow v$;               % state $w_1$ %
(4)           $WR \leftarrow wr$                           % transition to $w_0$ %
(5)      else $d[wr] \leftarrow (1 - d[wr])$;              % state $w_2$ %
(6)           $BUF[wr, d[wr]] \leftarrow v$;               % state $w_2$ %
(7)           $D[wr] \leftarrow d[wr]$                     % transition to $w_0$ %
(8)  end if;                                               % state $w_0$ %
(9)  return()                                              % state $w_0$ %
end operation.

operation R.read() is                                     % state $r_0$ %
(10) if $(WR \neq rr)$                                     % transition to $r_1$ or $r_2$ %
(11)     then $rr \leftarrow (1 - rr)$;                    % state $r_1$ %
(12)          $RR \leftarrow rr$                           % transition to $r_2$ %
(13) end if;                                               % state $r_2$ %
(14) let $d = D[rr]$;                                      % transition to $r_3$ %
(15) $res \leftarrow BUF[rr, d]$;                          % state $r_3$ %
(16) return($res$)                                         % state $r_3$ %
end operation.                                            % empty transition to $r_0$ %
```

Fig. 13.2 Tromp's construction of a SWSR b-valued atomic register

group $1 - RR$). So, the writer simply writes to a buffer in that other group. To do so, it first determines this new group (setting $wr \leftarrow 1 - wr$, line 2), then writes into the target buffer $BUF[wr, d[wr]]$ (line 3), and finally sets WR to its new value wr (line 4). The aim of this statement is also to have $WR \neq RR$ in order to indicate to the reader that a new value was written.

- $RR \neq WR$.
  In this case the current value of the first-level switch indicates that (if it is reading) the reader is currently reading from the other buffer group (namely the RR group). The writer writes accordingly in the same group as its previous write, but in the other buffer, namely $BUF[WR, D[1 - WR]]$ (lines 5–6). This is in order to prevent a possible collision in the case where the reader would terminate its read and start a new read accessing now the same buffer group WR. (This can occur due to asynchrony. As process speeds are independent, several read invocations can overlap the same write invocation as shown in Fig. 13.3.) The writer finally updates $D[wr]$ to have again $d[WR] = D[WR]$ (line 7). Let us observe that, in that case, the writer does not modify WR.

The algorithm implementing the operation R.read() The algorithm for the operation R.read() is described in lines 10–16 of Fig. 13.2.

As WR denotes the last group where a value was written, the reader first establishes $RR = WR$ before reading, in order to obtain the number of the buffer group containing the last value that was written (lines 10–13). As we have seen in the write algorithm, that value is kept in the buffer $BUF[WR, D[WR]]$. As the reader is the only process that

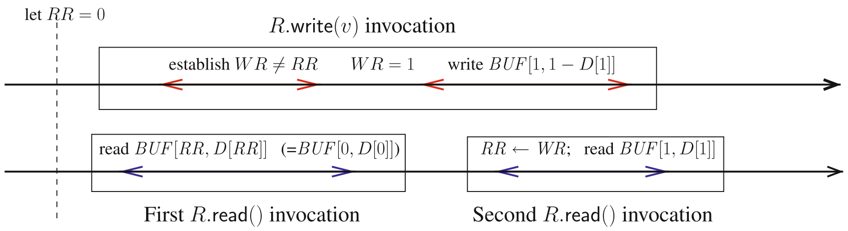

let $RR = 0$
R.write(v) invocation
establish $WR \neq RR$
$WR = 1$
write $BUF[1, 1 - D[1]]$
read $BUF[RR, D[RR]]$ (=$BUF[0, D[0]]$)
$RR \leftarrow WR$; read $BUF[1, D[1]]$
First R.read() invocation
Second R.read() invocation

Fig. 13.3 A write invocation concurrent with two read invocations

updates RR and we have $RR = WR$ after such an update, that buffer is unambiguously identified for the reader as $BUF[RR, D[RR]]$. The reader reads that buffer, and returns its value as the result of the read operation (lines 14–16).

13.2.4 Proof of the Construction: Collision-Freedom

The proof of Tromp's construction is made up of two parts. The first part (addressed in this section) consists in establishing the collision-freedom property, namely, that there is no concurrent write and read operation on the same buffer. The second part shows that the constructed register R is an atomic b-valued register.

From atomic bits to finite state automata Let us consider an execution at an abstraction level involving only the read and the write operations on the base atomic bits WR, RR, $D[0]$, and $D[1]$. As they are atomic, this execution is linearizable (all the read and write operations on these bits can be totally ordered, in such a way that this total order respects their real-time occurrence order and a read of a register obtains the last value written in this register).

Said in another way, the fact that the bits are atomic allows reasoning as if the operations on these bits were executed one at a time. This means that we can associate with each algorithm an automaton where a transition corresponds to a read or a write of an atomic bit. A state of the associated automaton is then the code of the algorithm contained between two successive transitions. The idea is then to compute the cross-product of these automata to analyze all the possible behaviors that can be produced by concurrent read and write operations. The global automaton obtained in that way will then be used to show the collision-freedom property.

Write automaton The automaton associated with the write operation is described in Fig. 13.4. It has three states (denoted w_0, w_1, and w_2) and four transitions. These states and transitions are described on the right part of Fig. 13.2. All the line numbers that follows refer to the algorithms described in Fig. 13.2.

Initially, or after it has completed a write operation, the writer is in the local state w_0. Then, according to the value it reads from RR (line 1), it executes one out of two possible transitions. The reading of RR such that $RR = WR$ constitutes the transition from w_0 to w_1,while the reading of RR such that $RR \neq WR$ constitutes the transition

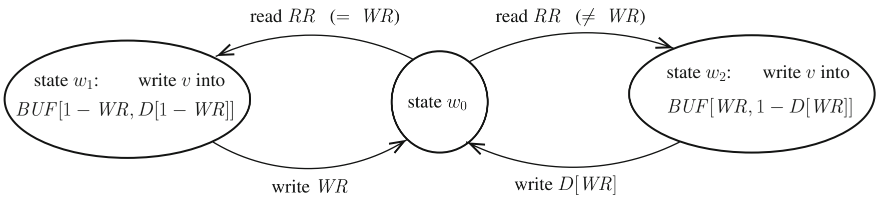

read RR (= WR)
read RR ($\neq$ WR)
state w_1: write v into $BUF[1 - WR, D[1 - WR]]$
state w_0
state w_2: write v into $BUF[WR, 1 - D[WR]]$
write WR
write $D[WR]$

Fig. 13.4 The write automaton of Tromp's construction

from w_0 to w_2. Then,when it is in the state w_1, the writer writes the safe bits of the buffer $BUF[1 - WR, D[1 - WR]]$ (line 3). When it is in the state w_2, it writes the safe bits of the buffer $BUF[WR, 1 - D[WR]]$ (line 6). Finally, the transition from w_1 to the initial state w_0 is the writing of the atomic bit WR (line 4), while the transition from w_2 to w_0 is the writing of the atomic bit $D[WR]$ (line 4).

Read automaton The automaton associated with the read operation is described in Fig. 13.5. It has four states (denoted r_0, r_1, r_2, and r_3) and four transitions.

Initially, the reader is in the local state r_0. Then, according to the value it reads from WR (this atomic read constitutes the first transition, line 10), it proceeds to the local state r_1 if $WR \neq RR$, or to the local state r_2 if $WR = RR$. When it is in state r_1, the reader modifies its local variable rr, and then executes the second transition, namely it updates RR to obtain $WR = RR$ (line 12), which makes it progress to the local state r_2 (line 13). Then, the atomic read of $D[RR]$ (line 14) constitutes its next transition which makes it proceed to state r_3. While it is in that state (lines 15–16) the reader reads the safe bits of the buffer whose coordinates have been previously determined, i.e., $BUF[RR, D[RR]]$. After, it has read that buffer, the reader returns to the local state r_0.

The composite automaton As each transition is an access to an atomic bit, and all these accesses are linearizable, a global state of the system can be represented (at this observation level) by the values of the atomic bits. More specifically, the behavior of the whole system can be represented by a global automaton describing all the possible global states and all the transitions (operations on an atomic bit) that make the system progress from one global state to another global state.

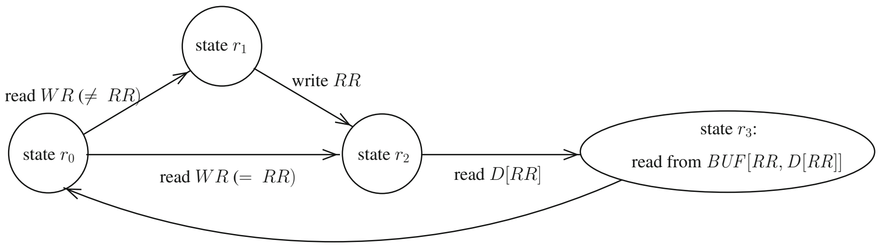


Fig. 13.5 The read automaton

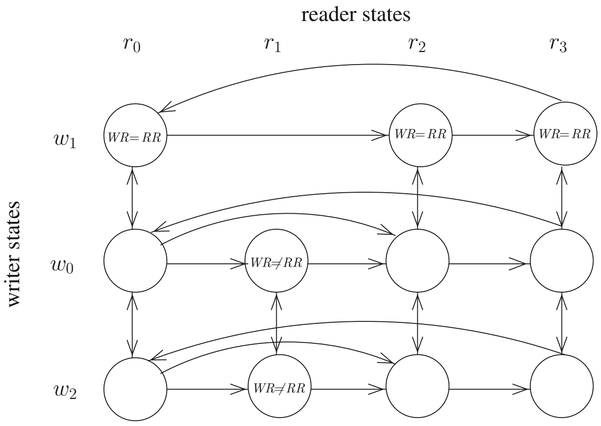


Fig. 13.6 Global state automaton

This global automaton can be easily obtained by computing the cross-product of the read and write automata. A global state is denoted $\langle w_i, r_j \rangle$, $0 \le i \le 2$, $0 \le j \le 3$, where w_i is a local state of the writer and r_j is a local state of the reader. As an example, $\langle w_2, r_3 \rangle$ denotes the state of the system where the writer is in the state w_2 and the reader is in the state r_3. This composite automaton, described in Fig. 13.6, is built as follows. The initial state is $\langle w_0, r_0 \rangle$. Then, each operation on an atomic bit that can be executed by the writer in state w_0 or the reader in state r_0 is considered. This determines four possible transitions from $\langle w_0, r_0 \rangle$. More explicitly, we have the following:

- If the writer atomically reads RR and $RR = WR$, it proceeds from w_0 to w_1, and consequently the system proceeds from $\langle w_0, r_0 \rangle$ to $\langle w_1, r_0 \rangle$.
- If the writer reads RR and $RR \neq WR$, the system proceeds from $\langle w_0, r_0 \rangle$ to $\langle w_2, r_0 \rangle$.
- If the reader reads WR and $RR = WR$, the system proceeds from $\langle w_0, r_0 \rangle$ to $\langle w_0, r_2 \rangle$.
- If the reader reads WR and $RR \neq WR$, the system proceeds from $\langle w_0, r_0 \rangle$ to $\langle w_0, r_1 \rangle$.

Then, this construction process has to be continued from each of the state that was obtained, i.e., from $\langle w_1, r_0 \rangle$, $\langle w_2, r_0 \rangle$, $\langle w_0, r_2 \rangle$, and $\langle w_0, r_1 \rangle$, and so on. Let us observe that, at the observation level defined by the w_i and r_j local states, there are at most 3×4 global states. The resulting global state automaton (Fig. 13.6) has 11 states.

It can be easily checked that the predicate $RR = WR$ defined on the system state is always true in the global states $\langle w_1, r_0 \rangle$, $\langle w_1, r_2 \rangle$ and $\langle w_1, r_3 \rangle$. Similarly, the predicate $RR \neq WR$ is always true in the global states $\langle w_0, r_1 \rangle$ and $\langle w_2, r_1 \rangle$. Let us also observe that the global state $\langle w_1, r_1 \rangle$ can never be reached during an execution. (It cannot be attained from $\langle w_1, r_0 \rangle$ because, $RR = WR$ being always true in $\langle w_1, r_0 \rangle$, the reader proceeds directly from r_0 to r_2 if it reads *WR*. Similarly, $\langle w_1, r_1\rangle$ cannot be attained from $\langle w_0, r_1\rangle$ as then a reading of *RR* by the writer obtains $RR \neq WR$, which entails the writer transition from w_0 to w_2.)

Lemma 27 *The write and read algorithms described in Fig. 13.2 ensure the collision-freedom property (i.e., if a process accesses a buffer, the other process does not concurrently access the same buffer).*

Proof The global state automaton shows that collisions can only occur in the global states $\langle w_1, r_3\rangle$ and $\langle w_2, r_3\rangle$. We examine each case separately:

- Global state $\langle w_1, r_3\rangle$.
  As indicated in Fig. 13.2, the writer is then accessing the buffer $BUF[1-WR, D[1-WR]]$ and the reader is accessing the buffer $BUF[RR, D[[RR]]$. As $RR = WR$ in the global state $\langle w_1, r_3\rangle$, it directly follows that the reader and the writer access buffers in distinct groups.
- Global state $\langle w_2, r_3\rangle$.
  The writer then accesses the buffer $BUF[WR, 1 - D[WR]]$, while the reader accesses the buffer $BUF[RR, D[[WR]]$. There are two sub-cases:
  - $RR \neq WR$. It follows directly that the reader and the writer access buffers in distinct groups.
  - $RR = WR$. The reader and the writer access buffers in the same group. The reader accesses then $BUF[RR, D[RR]]$ (line 14–15) while the writer accesses $BUF[RR, 1 - D[RR]]$. As $D[RR]$ and $1 - D[RR]$ are different values, it follows that the reader and the writer access distinct buffers. □

13.2.5 Correctness Proof

The previous lemma has shown that each buffer taken separately behaves as an atomic buffer (let us recall that a safe or regular register that is never accessed concurrently by the reader and the writer behaves as if it was atomic). Unfortunately, this observation is not sufficient to prove that the register R that is built is atomic. The buffer that is accessed by the reader could contain an old value that would make R non-atomic.

Theorem 55 *Tromp's construction described in Fig. 13.2 builds an SWSR b-valued atomic register.*

Proof To prove that R is atomic we use Theorem 53, stated and proved in Chap. 12.

Let r, r_1, and r_2 denote invocations of R.read(), and w denote an invocation of R.write(). Let us first observe that, thanks to Lemma 27, we can define a reading function $\pi()$ such that each read invocation r is mapped to the write invocation $\pi(\mathsf{r})$ that wrote (into some buffer $BUF[i, j]$) the value v read by r. Theorem 53 states that an execution history $\widehat{H}$ of a register R is atomic if the three following properties are satisfied:

- A0: $\forall\, \mathsf{r} : \neg\big(\mathsf{r} \rightarrow \pi(\mathsf{r})\big)$ ($\widehat{H}$ is an execution history).
- A1: $\forall\, \mathsf{r}, \mathsf{w} : \quad (\mathsf{w} \rightarrow_H \mathsf{r}) \Rightarrow \big((\pi(\mathsf{r}) = \mathsf{w}) \vee (\mathsf{w} \rightarrow_H \pi(\mathsf{r}))\big)$ (no read obtains an overwritten value).
- A2: $\forall\, \mathsf{r}_1, \mathsf{r}_2 : \ (\mathsf{r}_1 \rightarrow_H \mathsf{r}_2) \Rightarrow \big[\big(\pi(\mathsf{r}_1) = \pi(\mathsf{r}_2)\big) \vee \big(\pi(\mathsf{r}_1) \rightarrow_H \pi(\mathsf{r}_2)\big)\big]$ (no new/old inversion).

Proof of A0: $\forall\, \mathsf{r} : \neg\big(\mathsf{r} \rightarrow \pi(\mathsf{r})\big)$ ($\widehat{H}$ is an execution history). This is an immediate consequence of the definition of the function $\pi()$ given above. (Recall that the definition of $\pi()$ rests on Lemma 27.)

Proof of A1: $\forall\, \mathsf{r}, \mathsf{w} : \quad (\mathsf{w} \rightarrow_H \mathsf{r}) \Rightarrow \big((\pi(\mathsf{r}) = \mathsf{w}) \vee (\mathsf{w} \rightarrow_H \pi(\mathsf{r}))\big)$ (no read obtains an overwritten value).
The proof is by contradiction. Let us assume that there is a write invocation w such that $\pi(\mathsf{r}) \neq \mathsf{w}$ and $\pi(\mathsf{r}) \rightarrow_H \mathsf{w} \rightarrow_H \mathsf{r}$.

Let us assume without loss of generality the following context: (1) the operation w writes the buffer $BUF[0,0]$ and (2) $D[1] = 0$ when the event $resp[\mathsf{w}]$ occurs. (If $D[1] = 1$ when the event $resp[\mathsf{w}]$ occurs, the cases 1 and 2 remain unchanged in the case analysis that follows, while the cases 3 and 4 have to be exchanged.) It follows from this context and the write algorithm that, when $resp[\mathsf{w}]$ occurs, w has just written into $BUF[WR, D[WR]]$, which means that we then have $WR = 0$ and $D[0] = 0$. We consider the four possible buffers from which r can read:

1. The read invocation r reads from $BUF[0, 0]$.
   According to the definition of $\pi()$, this means that $\pi(\mathsf{r}) = \mathsf{w}$ or $\mathsf{w} \rightarrow_H \pi(\mathsf{r})$, contradicting the initial assumption. So, this case is eliminated.

2. The read invocation r reads from $BUF[0, 1]$.
   In this case (as the reader reads $BUF[RR, D[RR]]$), we have $RR = 0$ and $D[RR] = D[0] = 1$, from which we can conclude that the writer has changed $D[0]$ from 0 to 1 between the event $resp[\mathsf{w}]$ and the reading of $D[0]$ by the reader (notice that $D[0]$ is updated only by the writer). But, according to the write algorithm, this update of $D[0]$ is done at line 7, i.e., just after the writer has written in $BUF[0, 1]$, which implies that $\mathsf{w} \rightarrow_H \pi(\mathsf{r})$, leading to a contradiction which eliminates this case.
   The last two cases are similar. We only need to show that the buffer read by the read invocation r was written after the write invocation w.

3. The read invocation r reads from $BUF[1, 0]$.
   In this case, as the reader reads $BUF[RR, D[RR]]$, we have $RR = 1$.We can then conclude from lines 10 and 12 that the reader has previously read 1 from WR. This means that the writer has changed WR from 0 to 1. As this change can be done only at line 4, we conclude that the writer has previously written the buffer $BUF[WR, D[WR]]$, i.e., $BUF[1, 0]$. It follows that $w \rightarrow_H \pi(\mathsf{r})$, leading to a contradiction and eliminating this case.

4. The read invocation r reads from $BUF[1, 1]$.
   In this case, we have $RR = 1$ and the reader reads $D[RR] = D[1] = 1$. As we have $D[1] = 0$ when the writer terminates the invocation w (assumption on the value of $D[1]$ when the event $resp[\mathsf{w}]$ occurs), we can conclude that the writer has changed $D[1]$ from 0 to 1. As this can occur only at line 7, we conclude that the writer has previously written the buffer $BUF[1, D[1]]$, i.e., $BUF[1, 1]$.As in the previous cases, it follows that $\mathsf{w} \rightarrow_H \pi(\mathsf{r})$, which leads to a contradiction.

It follows from this case analysis that the read invocation r obtains a value written by an invocation $\pi(\mathsf{r})$ such that $\mathsf{w} \rightarrow_H \pi(\mathsf{r})$, which proves that the history $\widehat{H}$ satisfies property $A1$.

Proof of A2: $\forall\, \mathsf{r}_1, \mathsf{r}_2 : (\mathsf{r}_1 \rightarrow_H \mathsf{r}_2) \Rightarrow [\big(\pi(\mathsf{r}_1) = \pi(\mathsf{r}_2)\big) \vee \big(\pi(\mathsf{r}_1) \rightarrow_H \pi(\mathsf{r}_2)\big)]$ (no new/old inversion.) We prove that the history $\widehat{H}$ satisfies the property A2 by showing that a violation of A2 would imply a violation of A1.

Let r_1 and r_2 be two read invocations such that $\pi(\mathsf{r}_2) \rightarrow_H \pi(\mathsf{r}_1)$. It follows from the very definition of $\pi()$ that the operation r_1 accesses some buffer (let us say $BUF[0, 0]$) after $\pi(\mathsf{r}_1)$ has written into it. But, as the write invocation $\pi(\mathsf{r}_1)$ terminates by changing an atomic bit (line 4 or line 7), we can conclude that $resp[\pi(\mathsf{r}_1)] <_S inv[\mathsf{r}_1] <_S resp[\mathsf{r}_1] <_S inv[\mathsf{r}_2]$, where $\widehat{S}$ is a linearization of $\widehat{H}$ at the level of the atomic bits. We consequently have $\pi(\mathsf{r}_2) \rightarrow_H \pi(\mathsf{r}_1) \rightarrow_H \mathsf{r}_2$, which violates property A1 and terminates the proof. $\square$

13.3 A Construction Based on Impure Buffers

This section presents K. Vidyasankar's construction (1990) of an atomic b-valued register which is based on impure buffers. This means that (differently from the previous construction) this construction does not systematically prevent the reader and the writer from simultaneously accessing the same buffer. When this occurs, the value read from the corresponding buffer can be arbitrary. When there is a collision in a buffer, the construction directs the colliding read invocation to read the value of another buffer.

13.3.1 Internal Representation of the Atomic b-Valued Register R

As already indicated, the construction uses three SWSR b-valued buffers each made up of $\lceil \log_2 b \rceil$ safe bits and three SWSR atomic bits that implement a switch mechanism:

- The three b-valued buffers, denoted $BUF[0]$, $BUF[1]$, and $HELP\_BUF$, are written by the writer and read by the reader. They are initialized to the initial value of the constructed register R.

- The three SWSR atomic bits implementing the switch are denoted *RR*, *WR*, and *LAST* (they are all initialized to 0). *RR* is written by the reader, while *WR* and *LAST* are written by the writer.

13.3.2 An Incremental Construction

Underlying principle The idea of the construction is the following.Consecutive invocations of R.write() issued by the writer alternately write *BUF*[0], *BUF*[1], *BUF*[0], *BUF*[1], etc. Moreover, the number of the the last buffer that was written is kept in the atomic bit *LAST*.

The idea is then for the reader to read from *BUF*[*LAST*], and for the writer to first write into *BUF*[1 − *LAST*] and then change the value of *LAST* so that it points to the last written buffer.

Unfortunately, as shown in Fig. 13.7, this is not sufficient to prevent the reader and the writer from simultaneously accessing the same buffer. This collision problem can be solved in two steps.

- First, the atomic bits *WR* and *RR* are used to implement a handshaking mechanism that allows the reader to know whether there was a possible collision in the buffer *BUF*[*LAST*] it has just read. If there is no collision, the value read from *BUF*[*LAST*] is returned.
- If there is a possibility of collision, the third buffer *HELP_BUF* is used to provide the reader with a value that it may return.

Incremental construction The way the atomic bits *WR* and *RR* and the b-valued buffer *HELP_BUF* are managed constitutes the core of the construction. We present their management in an incremental way. The first step focuses on the management of *WR* and *RR*, while the second step focuses on the management of *HELP_BUF*.

To simplify the presentation, we allow the reader to read *RR* and the writer to read *WR* and *LAST*. (This is done without loss of generality, as a process always knows the exact value of an atomic bit for which it is the only writer.)

A first step towards the read and write algorithms As indicated previously, the writer first writes the new value v in the buffer *BUF*[1 − *LAST*] and then changes accordingly the value of *LAST*. Moreover, it indicates that a new value was written by setting *WR* = *RR*. This corresponds to the following sequence of statements:

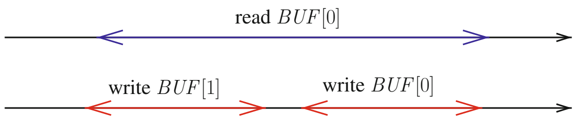


Fig. 13.7 Simultaneous accesses to the same buffer

```
1 $BUF[1 - LAST] \leftarrow v$;
2 $LAST \leftarrow (1 - LAST)$;
3 $r \leftarrow RR$;
4 **if** $WR \neq r$ **then** $WR \leftarrow r$ **end if**.
```

The reader first sets *RR* different from *WR* to indicate that a read is currently executing, and reads $BUF[LAST]$. Then it checks for a possible collision. To attain this goal, a "no-collision" predicate has to be defined. This predicate has to always answer true when there is a collision (moreover,it can be conservative in the sense that it may answer false in some cases where there is no collision).

Such a no-collision predicate can be defined from *RR* and *WR*. More specifically, it is $RR \neq WR$. It follows intuitively from the following observation. When it is true, it means that, since the time the reader has set *RR* different from *WR* (line 1 of the read operation described below), the writer has not accessed *WR*, otherwise we would have $WR = RR$ (see line 4 of the write operation). So, when the predicate is true, between line 1 and line 4 of the read operation, *LAST* contains the number of the last buffer that was written (its value has then been determined at line 2 of the write operation). The proof will show that this intuition is correct.

Let us observe that, if the predicate is false, we can conclude that a write operation is concurrent with the current read. Let us also notice that the "no-collision" predicate is not a "no-concurrency" predicate: a write operation can be concurrent with a read operation that evaluates the predicate to true. We finally obtain the read operation described below:

- If $RR \neq WR$, the predicate indicates that there was no collision on $BUF[LAST]$. The value of $BUF[LAST]$ is then saved in a local variable denoted $prev$ and returned. This no-collision case is indicated by the bit 1 returned with the value.
- If $RR = WR$, a collision was possible. In this case, the returned value is the most recent value whose reading was detected as being collision-free, i.e., the value $prev$. This is indicated by the bit 0 returned with the value.

```
1 $RR \leftarrow (1 - WR)$;
2 $val \leftarrow BUF[LAST]$;
3 $wr \leftarrow WR$;
4 **if** $RR \neq wr$ **then** $prev \leftarrow val$; return$(val, 1)$
5                   **else** return$(prev, 0)$
6 **end if**.
```

It is easy to show that the register *R* built by the previous read and write algorithms satisfies the following properties:

- It is safe: an invocation of R.read() executed without concurrent write invocations returns the last value that was written in R.
- The value returned by an invocation of R.read() is a value that was written. It follows that the register is more than a safe register, as it never returns an arbitrary value in the presence of concurrency.
- Due to the use of the $prev$ local variable, a returned value is never older than the previously returned value. In this sense, the previous construction prevents new/old inversions.

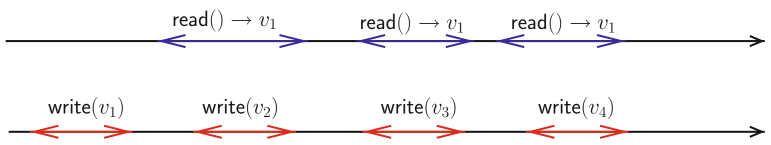


Fig. 13.8 Successive read/write collisions

But unfortunately this construction does not provide an atomic register R. It does detect collision and prevent new/old inversion, but an invocation of R.read() can return an older value than the last value written before that read invocation. This can occur when consecutive read invocations detect possible collisions as depicted in Fig. 13.8. The first read invocation sets $prev$ to $v1$, and then, as each of the following read invocations detects a collision, it returns the current value of $prev$ without modifying it.

13.3.3 The Algorithms Implementing the Operations R.write() *and* R.read()

The algorithm implementing the operation R.write() The simple addition of a third buffer (denoted *HELP_BUF*) allows an atomic register R to be obtained. The final R.write(v) operation is described in Fig. 13.9 (the comments refer to the proof). The only modification with respect to the previous write algorithm consists in the addition of the statement $HELP\_BUF \leftarrow v$ at line 4. When there are only write invocations, we permanently have $WR = RR$. So, the writer can suspect a possible collision when it finds $WR \neq RR$. When this occurs, the writer helps the reader by writing the value v in the additional buffer *HELP_BUF*.

The algorithm implementing the operation R.read() The final algorithm implementing R.read() is described in Fig. 13.9. The only modifications with respect to the previous tentative solution consist in suppressing the local variable $prev$, and returning instead the value of the additional buffer *HELP_BUF* when a possible collision in $BUF[LAST]$ is detected (i.e., when $RR \neq WR$). The proof will show that there is no collision in *HELP_BUF* (i.e., it is never written when it is read).

13.3.4 Proof of the Construction

Notations The proof uses the following notations:

- The notation r is used to denote an invocation of R.read(), w an invocation of R.write(), and op an invocation of R.read() or R.write().

```
operation R.write(v) is
(1)   $BUF[1-LAST] \leftarrow v$;                          % $w$-read-$BUF[1-LAST]$ %
(2)   $LAST \leftarrow (1-LAST)$;                          % $w$-write-$LAST$ %
(3)   $rr \leftarrow RR$;                                  % $w$-read-$RR$ %
(4)   if $WR \neq rr$ then $HELP\_BUF \leftarrow v$;        % $w$-write-$HELP\_BUF$ %
(5)                        $WR \leftarrow rr$               % $w$-write-$WR$ %
(6)   end if
(7)   return()
end operation.

operation R.read() is
(8)   $RR \leftarrow (1-WR)$;                % $r$-1read-$WR$ followed by $r$-write-$RR$ %
(9)   $val \leftarrow BUF[LAST]$;            % $r$-read-$LAST$ followed by $r$-read-$BUF[LAST]$ %
(10)  $wr \leftarrow WR$;                    % $r$-2read-$WR$ %
(11)  if $RR \neq wr$ then return($val$)
(12)            else return($HELP\_BUF$)     % $r$-read-$HELP\_BUF$ %
(13)  end if
end operation.
```

Fig. 13.9 Vidyasankar's construction of an SWSR b-valued atomic register

- The notation op-read-X (op-write-X) denotes the invocation of a read (write) operation on the base register X issued by op.
- $\widehat{H}$ denotes an execution history involving all the operation invocations on the base objects. More specifically, those are the read and write invocations on the atomic bits WR, RR, and $LAST$, and on the safe b-valued buffers $BUF[0]$, $BUF[1]$, and $HELP\_BUF$; $\rightarrow_H$ denotes the associated partial order.
- As the bits WR, RR, and $LAST$ are atomic, the sub-history of $\widehat{H}$ involving only the read and write invocations on these base registers is linearizable. Let $\rightarrow_{S\_ab}$ denote the corresponding total order on these invocations (the subscript "ab" stands for "atomic bit"; it is used to stress the fact the total order is defined only on the atomic bits and not on the safe b-valued registers).
- Each invocation r of R.read() reads the atomic bit WR twice (lines 8 and 10 of Fig. 13.9). The first of these reads is denoted r-1read-WR and the second is denoted r-2read-WR.
- Given an invocation w of R.write(v), we say that the base write operation w-write-WR occurs in between r-1read-WR and r-2read-WR if r-1read-WR $\rightarrow_{S\_ab}$ w-write-WR $\rightarrow_{S\_ab}$ r-2read-WR.

Lemma 28 *Let* inv$_1$ *and* inv$_2$ *be be two invocations of read or write operations on base atomic bits. We have* (inv$_1$ $\rightarrow_H$ inv$_2$) $\Rightarrow$ (inv$_1$ $\rightarrow_{S\_ab}$ inv$_2$).

Proof Let $\rightarrow_{H\_ab}$ be the sub-history of $\rightarrow_H$ involving only the operations on the atomic bits. The lemma then follows directly from the definition of the linearization order, which states that $\rightarrow_{H\_ab} \subseteq \rightarrow_{S\_ab}$. □

Lemma 29 *Each time the atomic bit WR is written, its new value is the complement of its previous value.*

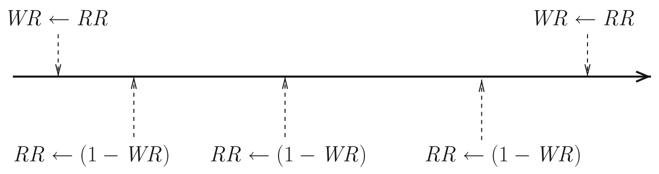


Fig. 13.10 Successive updates of the atomic bit *WR*

Proof Let us first observe that the atomic bit *RR* is read only by the writer and written only by the reader. It is always updated to $1 - WR$ (line 8) where the reader establishes $RR \neq WR$ each time it starts reading. (Let us notice that, if *WR* has not been modified since its last reading at line 8, this update of *RR* does not modify its value.)

The atomic bit *WR* is read only by the reader and written only by the writer. Its new value is then the value of *RR* (lines 3–5). More explicitly, the writer always establishes $WR = RR$ when it terminates a write invocation. It follows that *WR* takes then the value $RR = 1 - WR'$, where WR' denotes its previous value (see Fig. 13.10). □

Lemma 30 *Let* a, b, c *and* d *be four operation invocations of* $\widehat{H}$ *such that* b *and* c *are invocations on an atomic register. We have* $(\mathsf{a} \rightarrow_H \mathsf{b} \rightarrow_{S\_ab} \mathsf{c} \rightarrow_H \mathsf{d}) \Rightarrow (\mathsf{a} \rightarrow_H \mathsf{d})$.

Proof Due to Lemma 28, we can conclude from $\mathsf{b} \rightarrow_{S\_ab} \mathsf{c}$ that either $\mathsf{b} \rightarrow_H \mathsf{c}$ or the invocations b and c are concurrent in $\widehat{H}$. This means that (a) the event $inv[\mathsf{b}]$ occurs before the event $resp[\mathsf{c}]$ (c terminates before b starts, see Fig. 13.11). As $\mathsf{a} \rightarrow_H \mathsf{b}$, the event $inv[\mathsf{b}]$ occurs after the event $resp[\mathsf{a}]$ (b). Similarly, as $\mathsf{c} \rightarrow_H \mathsf{d}$, the event $resp[\mathsf{c}]$ occurs before the event $inv[\mathsf{d}]$ (c).

Combining (a), (b), and (c), we conclude that the event $resp[\mathsf{a}]$ occurs before the event $inv[\mathsf{d}]$, i.e., $\mathsf{a} \rightarrow_H \mathsf{d}$. □

Lemma 31 *Let* w *be an invocation of* R.write() *that writes WR. This invocation* w *sets* $WR = RR$. *Moreover, if there is an invocation* r *of* R.read() *such that the base operation* w*-write-WR occurs in between* r*-1read-WR and* r*-2read-WR, then the equality* $WR = RR$ *continues to hold until* r *terminates.*

Proof Let w_i ($i \geq 1$) be the ith invocation of R.write() that writes the atomic bit *WR*. The proof is by induction. It uses the fact that the base operations which are concerned are atomic bits (and we can consequently base our reasoning on the associated linearization order).

Let us first consider w_1. Due to Lemma 29 and the fact that *WR* is initialized to 0, we conclude that w_1 writes 1 into *WR*. It has consequently read 1 from *RR*. As *RR* is initialized to 0, it follows that there is an invocation of R.read() that has written

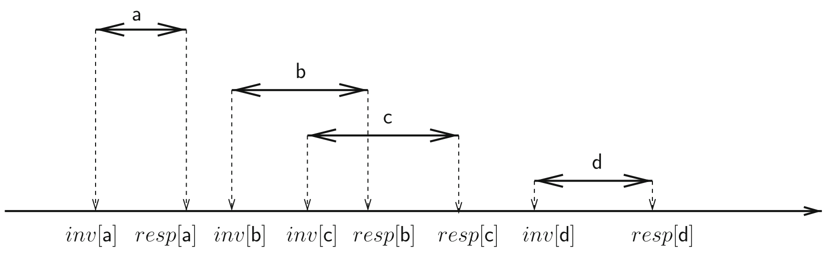


Fig. 13.11 Ordering on base operations (Lemma 30)

1 into *RR*. Let r_1 be the first of these read operations (in case there are several such read invocations). It follows from the previous observation that r_1-write-*RR* $\rightarrow_{S\_ab}$ w_1-read-*RR*. Let us also notice that, as the base operation r_1-write-*RR* writes 1 in *RR*, we can conclude that the base operation w_1-1read-*WR* reads 0 from *WR*.

Let us consider any subsequent invocation r′ of *R*.read() (if any) that performs r′-1read-*WR* before w_1-write-*WR* (i.e., such that r′-1read-*WR* $\rightarrow_{S\_ab}$ w_1-write-*WR*). Due to Lemma 29 and to the fact that the base invocation w_1-write-*WR* writes 1 in *WR*, we can conclude that such a read invocation r′ reads also 0 from *WR* and consequently the base operation r′-write-*RR* writes 1 in *RR*. (Let us observe that this is irrespective of the linearization order of the base operations r'-write-*RR* and w_1-read-*RR*.) This means that these read invocations r′ write in *RR* the same value as the read invocation r, namely the value 1.

It follows that, after w_1 has executed w_1-write-*WR*, we have $RR = WR = 1$. Moreover, if w_1-write-*WR* occurs in between r-1read-*WR* and r-2read-*WR* of some read operation r, then the equality $WR = RR$ continues to hold until the next read operation that writes *RR*, i.e., at least until r terminates (this follows from the fact that any other write operation does not modify *WR* because we have then $WR = RR$). This completes the proof for the base case, namely for the first write invocation (w_1) which writes *WR*.

Assuming as the induction hypothesis that the assertion holds from w_1 until w_i. The same reasoning as the previous one shows that the assertion also holds for w_{i+1}. □

Lemma 32 *Let* r *be an invocation of R*.read(). *(a) There is at most one write invocation* w *such that* w*-write-WR occurs in between* r*-1read-WR and* r*-2read-WR. (b) If there is no invocation* w *of R*.write() *such that* w*-write-WR occurs in between* r*-1read-WR and* r*-2read-WR, then the value v returned by* r *is a value obtained from BUF*[0] *or BUF*[1]. *If there is such a write invocation* w*, then the value v returned by* r *is a value obtained from HELP_BUF, and that safe register was written by* w.

Proof The proof of (a) is an immediate consequence of Lemma 31 that states that, if there is a write invocation w such that w-write-*WR* occurs in between r-1read-*WR* and r-2read-*WR*, then the equality $WR = RR$ holds until the end of the read operation r. As $WR = RR$ until the end of r, it follows from the test of line 4 that no write invocation w′ can write *WR* until the end of r, which proves item (a).

The proof of item (b) follows directly from the code of the read algorithm. If *WR* is not modified between r-1read-*WR* and r-2read-*WR*, these base read invocations return the same value from *WR*. It then follows from line 11 that the value returned by the read operation r comes from $BUF[0]$ or $BUF[1]$. If *WR* is modified between r-1read-*WR* and r-2read-*WR*, it follows from (a) that there is at most one write invocation w that writes *WR*, and the base read invocations r-1read-*WR* and r-2read-*WR* return different values from *WR*. It follows from the test of line 11 that the value returned by the read invocation r comes from *HELP_BUF*. Moreover, due to lines 4–5, the invocation w has written into *HELP_BUF* before it writes *WR*, which concludes the proof of the lemma. □

The next lemma shows that, although the b-valued buffers $BUF[0]$, $BUF[1]$, and *HELP_BUF* are only safe registers, if one of their values is returned by an invocation r of R.read(), that value is a value that was written by an invocation w of R.write().

Lemma 33 *Let B be the b-valued base buffer whose value was returned by an invocation* r *of* R.read() *(B is $BUF[0]$, $BUF[1]$, or HELP_BUF). When the reader was reading B, the writer was not writing B.*

Proof We consider two cases. Let us first examine the case where there is a write invocation w such that w-write-*WR* occurs in between r-1read-*WR* and r-2read-*WR*.

Due to item (b) of Lemma 32, r returns a value read from the base b-valued buffer *HELP_BUF* that was written by w. Let us notice that, due to the code of the write algorithm, we have w-write-*HELP_BUF* $\rightarrow_H$ w-write-*WR*. Similarly, due to the code of the read algorithm, we have r-2read-*WR* $\rightarrow_H$ r-read-*HELP_BUF*. Moreover, we also have w-write-*WR* $\rightarrow_{S\_ab}$ r-2read-*WR* (case assumption). Combining these relations we obtain w-write-*HELP_BUF* $\rightarrow_H$ w-write-*WR* $\rightarrow_{S\_ab}$ r-2read-*WR* $\rightarrow_H$ r-read-*HELP_BUF*.

Using now Lemma 30, we obtain w-write-*HELP_BUF* $\rightarrow_H$ r-read-*HELP_BUF*, which shows that the writing of *HELP_BUF* by w and its reading by r are not concurrent. Finally, as the equality $RR = WR$ holds until r terminates (Lemma 31), no subsequent write invocation w$'$ writes *HELP_BUF* until r completes, i.e., until it has finished reading *HELP_BUF*.

Let us now examine the case where there is no invocation w of R.write() such that w-write-*WR* occurs in between r-1read-*WR* and r-2read-*WR*. Due to Lemma 32, the read operation r returns a value that it has obtained by reading $BUF[0]$ or $BUF[1]$. Without loss of generality let us assume that $LAST = 0$ when r reads it (line 9). This means that r reads from the safe b-valued register $BUF[0]$.

Let us observe that $BUF[0]$ either contains the initial value or was previously written by some write operation w_0; this is because a write operation first writes $BUF[0]$ and only then updates *LAST* to 0 (lines 1–2), that is, to the value of *LAST* subsequently read by r. If there is no other write operation, r reads $BUF[0]$ after it was written and consequently the writer is not writing $BUF[0]$ when the reader is reading it, which proves the lemma.

If there is a next write invocation w′, it writes 1 in *LAST* (line 2) and does so after r has read *LAST* (i.e., r-read-*LAST* $\rightarrow_{S\_ab}$ w′-write-*LAST*), otherwise r would have read the value 1 from *LAST*. This means that w′ writes *BUF*[1] whereas r reads from *BUF*[0]. Furthermore, as r returns the value read from *BUF*[0], the invocation r-2read-*WR* was such that $WR \neq RR$ (line 11). Consequently, w′ finds $WR \neq RR$ when it reads *RR*, and hence writes *WR* (lines 4 – 5). As r-1read-*WR* $\rightarrow_H$ r-read-*LAST*, w′-write-*LAST* $\rightarrow_H$ w′-write-*WR* and r-read-*LAST* $\rightarrow_{S\_ab}$ w′-write-*LAST*, it follows from Lemma 30 that r-1read-*WR* $\rightarrow_H$ w′-write-*WR*; due to the case assumption, we also have r-2read-*WR* $\rightarrow_H$ w′-write-*WR*. This means that no write invocation starting after w′ starts executing before r-2read-*WR*. It follows that, when r was reading from *BUF*[0] (i.e., between its two base read invocations on *WR*, namely r-1read-*WR* and r-2read-*WR*), no write invocation was writing *BUF*[0], which proves the case and completes the proof. □

Theorem 56 *Vidyasankar's construction described in Fig. 13.9 builds an SWSR b-valued atomic register R.*

Proof The proof is incremental. It first shows that *R* is safe, then it shows it is regular, and finally atomic.

R is safe. A read invocation r which is not concurrent with a write invocation finds always $RR \neq WR$ (line 4). This is because there is no write into *WR* between r-1read-*WR* and r-2read-*WR*. Consequently such a read invocation r returns the value of *BUF*[0] or *BUF*[1]. Moreover, as there is no concurrent write invocation, *LAST* points to the buffer entry containing the last written value. It then follows that the value returned by the read invocation r is the last written value. The register *R* is consequently safe.

R is regular. Given an invocation r of *R*.read(), let π(r) be the invocation of *R*.write() that wrote into a buffer the value read by r. (Due to Lemma 33, the value that is read was written; despite the fact the *b*-valued buffers are only safe, it is not an arbitrary value.) We have from the construction that $\neg(\pi(\mathsf{r}) \rightarrow_H r)$, i.e., π(r) precedes or is concurrent with *r*. Moreover, let w denote the last write invocation completed before r started; without loss of generality, let us assume that w wrote *BUF*[0]. We consider three cases:

- The value returned by r comes from *BUF*[0]. That value has then been written by w or a later write w′ that is concurrent with r (that write is such that it set *LAST* to 0). So, π(r) is w or a later write concurrent with r. It follows that *R* is regular.
- The value returned by r comes from *BUF*[1]. In this case, a successor w′ of w that overlaps r must have written 1 in *LAST*. According to lines 1–2, w′ has previously written *BUF*[1]. Moreover, the base w′-read-*LAST* invocation obtains the value 1 from *LAST* (we have w′-write-*LAST* $\rightarrow_{S\_ab}$ r-read-*LAST*), and then reads *BUF*[1]. It follows that π(r) is w′ or a successor of w′ that wrote 1 into *LAST* and is concurrent with r. It follows that *R* is regular.

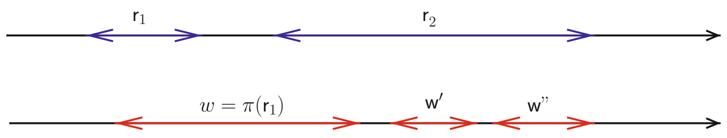


Fig. 13.12 Overlapping invocations (atomicity in Theorem 56)

- The value returned by r comes from *HELP_BUF*. In that case, due to the second item of Lemma 32, there is an invocation w' of R.write() whose base invocation w'-write-*WR* occurs between r-1read-*WR* and r-2read-*WR*, from which we conclude that w' and r are concurrent. Moreover, due to the last part of Lemma 32 and Lemma 33, the value returned by r is the value written by w'. It follows that $\pi(\mathsf{r}) = \mathsf{w}'$, which proves the regularity of R.

R is atomic. As R is regular, we show it is atomic by proving that there is no new/old inversion. (The atomicity follows then directly from Theorem 43 of Chap. 11 that proved that an atomic register is a regular register with no new/old inversion.)

Let r_1 and r_2 be two invocations of R.read() such that $\mathsf{r}_1 \rightarrow_H \mathsf{r}_2$, and let $\pi(\mathsf{r}_1) = \mathsf{w}$. We show that $\pi(\mathsf{r}_2)$ is either w or a later write.

Let us first observe that, if r_2 is not concurrent with w, due to the regularity of R, $\pi(\mathsf{r}_2)$ is either w or a later write. So, we assume in the following that r_2 is concurrent with w. Let us notice that r_1 is then also concurrent with w, otherwise we would have $\mathsf{w} \rightarrow_H \mathsf{r}_1 \rightarrow_H \mathsf{r}_2$, contradicting the fact that w and r_2 are concurrent (Fig. 13.12). Moreover, if r_2 is concurrent with several write operations, w is the first of them. As previously, let us assume without loss of generality that w writes *BUF*[0].

- If r_1 returns the value from *BUF*[0], we have from the code of the algorithms and the atomicity of *LAST*: w-write-*BUF*[0] $\rightarrow_H$ w-write-*LAST* $\rightarrow_{S\_ab}$ r_1-read-*LAST* $\rightarrow_H$ r_1-read-*BUF*[0].

- If r_1 returns the value from *HELP_BUF*, we have from the code of the algorithms, the atomicity of *LAST* and *WR*, and Lemma 32: w-write-*LAST* $\rightarrow_H$ w-write-*HELP_BUF* $\rightarrow_H$ w-write-*WR* $\rightarrow_{S\_ab}$ r_1-2read-*WR* $\rightarrow_H$ r_1-read-*HELP_BUF*.

As $\mathsf{r}_1 \rightarrow_H \mathsf{r}_2$, we have w-write-*LAST* $\rightarrow_{S\_ab}$ r_2-read-*LAST* in both cases.

The write invocation w writes 0 in *LAST* (assumption), but the read invocation r_2 may read 0 or 1 from the atomic bit *WR* (this can occur when r_2 is concurrent with other write operations that are issued after w). Whatever the value (0 or 1) returned by r_2-read-*LAST*, due to w-write-*LAST* $\rightarrow_{S\_ab}$ r_2-read-*LAST*, we have the following:

- If r_2 returns a value read from *BUF*[0] or *BUF*[1], that value has necessarily been written by w or a successor of w, which proves the case.

- If r_2 returns the value read from *HELP_BUF*, due to the second item of Lemma 32, that value was written by a write invocation w' such that the base invocation w'-write-*WR* is between r_2-1read-*WR* and r_2-2read-*WR*. Since w is the first write that is concurrent with r, it follows that w' is w or a subsequent write invocation w''. This completes the proof of the atomicity of R. □

```
operation $R$.write($v$) is
    $BUF[1 - LAST] \leftarrow v$;
    $LAST \leftarrow 1 - LAST$;
    for each $j \in \{1, \ldots, n\}$ do
        $rr \leftarrow RR[j]$;
        if $WR[j] \neq rr$ then $HELP\_BUF[j] \leftarrow v$;
                                $WR[j] \leftarrow rr$
        end if
    end for;
    return()
end operation.

operation $R$.read() is    % code for $p_i$ %
    $RR[i] \leftarrow 1 - WR[i]$;
    $val \leftarrow BUF[LAST]$;
    $wr \leftarrow WR[i]$;
    if $RR[i] \neq wr$ then return($val$)
                        else return($HELP\_BUF[i]$)
    end if
end operation.
```

Fig. 13.13 Vidyasankar's construction of an SWMR b-valued atomic register

13.3.5 From SWSR to SWMR b-Valued Atomic Register

In a very interesting way, the previous construction that builds an SWSR atomic b-valued register R can be easily extended to build an SWMR atomic register R. This simple construction is as follows.

The base b-valued safe buffers $BUF[0]$ and $BUF[1]$, together with the atomic bit $LAST$, are used exactly as in the basic algorithms described in Fig. 13.9. The only difference lies in the fact that now the writer considers explicitly the fact there are n readers $p_1, \ldots, p_n$ and executes the base write algorithm with respect to each of them.

To that end, the control bits WR and RR and the additional b-valued safe buffer $HELP\_BUF$ are replaced by the arrays $WR[1..n]$, $RR[1..n]$, and $HELP\_BUF[1..n]$, each with one entry per reader process p_i. More specifically, for each couple made up of the writer and a reader process p_i, the base registers $WR[i]$, $RR[i]$, and $HELP\_BUF[i]$ replace the base registers WR, RR, and $HELP\_BUF$ used in the basic SWSR construction.

The resulting SWMR construction is described in Fig. 13.13. It uses $2n + 1$ atomic bits ($WR[1..n]$, $RR[1..n]$, and $LAST$) and $n + 2$ b-valued safe buffers ($HELP\_BUF[1..n]$, $BUF[0]$, and $BUF[1]$) whose size is $\lceil \log_2 b \rceil$ bits.The cost of a read operation is the same as in the basic SWSR construction, while the cost of a write operation now depends on n, the number of readers. The proof of this SWMR construction is the same as the proof of the basic SWSR construction.

13.4 Summary

This chapter has presented two efficient constructions that build an SWSR b-valued atomic register from a few atomic bits and safe buffers of $\lceil \log_2 b \rceil$ bits. The atomic bits are used to implement switches directing the writer and the reader to an appropriate buffer to read or write a value.

These constructions differ in their underlying principles. The first one, due to J. Tromp, ensures that the reader and the writer never access the same buffer simultaneously. The second one, due to K. Vidyasankar, allows conflicts in a buffer but can direct the reader or the writer to sequentially access two buffers.

13.5 Bibliographic Notes

- The notions of safe register, regular register, and atomic register are due to L. Lamport [189, 190] who has also presented (in these papers) a suite of algorithms that allow the construction of an MWMR b-valued atomic register from SWSR safe bits.
- As already indicated, the first construction presented in this chapter is due to J. Tromp [265].
- As indicated, the second construction presented in this chapter is due to K. Vidyasankar [270]. Its starting point is a construction due to G.L. Peterson [225].
- Numerous other papers have presented constructions of SWSR b-valued atomic registers from "lower-level" registers, e.g., [52, 59, 72, 132, 133, 168, 177, 178, 184, 190, 196, 219, 225, 257, 269, 272] to cite a few.

Part VI
On the Foundations Side: The Computability Power of Concurrent Objects (Consensus)

The notion of a mutex-free implementation of an object and associated liveness properties (namely obstruction-freedom, non-blocking, and wait-freedom) were introduced in Part III (Chaps. 5–9). Wait-free implementations of a splitter, weak counter, store-collect, snapshot, and renaming objects were presented in these chapters. Some of these wait-free implementations rest on atomic read/write registers only, while others rest on atomic read/write registers plus more sophisticated objects such as compare&swap objects.

This part of the book addresses a more general issue, which is the fundamental question in the study of wait-free implementations of concurrent objects, namely

> Given concurrent objects of some type T_a and a concurrent atomic object Z (i.e., Z is defined by a sequential specification on total operations), is there a wait-free implementation of Z from atomic read/write registers and any number of objects of type T_a?

After having introduced the notion of a *universal object* and a *universal construction*, this part of the book presents the notion of a consensus object and two universal constructions based on consensus objects. It then shows that a consensus object cannot be wait-free implemented from atomic read/write registers only and introduces the notion of a *consensus number* for concurrent objects. Consensus numbers allow for the ranking of the computability power of concurrent objects in asynchronous systems in which any number of processes may crash. Finally, additional assumptions (such as synchrony assumptions or failure detectors) that allow consensus to be solved and associated algorithms building consensus objects are presented.

Chapter 14
Universality of Consensus

This chapter introduces the notion of a universal object and a universal construction and shows that the consensus object is universal. To that end two consensus-based universal constructions (which rest on different principles) are presented. This chapter shows also that binary consensus is as powerful as multi-valued consensus, hence binary consensus is universal.

Keywords Atomic object · Binary consensus · Bounded construction · Consensus object · Deterministic versus non-deterministic object · Linearizability · Multi-valued consensus · Sequential specification · Total operation · Total order · Universal construction · Universal object · Wait-freedom

14.1 Universal Object, Universal Construction, and Consensus Object

The universality notions developed and used in this chapter concern the synchronization power of concurrent objects in the presence of asynchrony and any number of process crashes. Hence, they address the computability power of concurrent objects.

14.1.1 Universal (Synchronization) Object and Universal Construction

Reminder on total operations and wait-freedom Atomic objects (also called linearizable objects) are defined by a sequential specification on total operations. An object operation is *total* if it can be applied to any state of the object. It follows that the invocation of a total operation can always be completed by the invoking process (i.e., whatever the behavior of the other processes). This important property, which

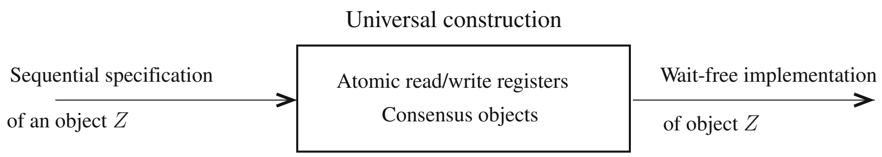

Universal construction
Sequential specification of an object Z
Atomic read/write registers
Consensus objects
Wait-free implementation of object Z

Fig. 14.1 From a sequential specification to a wait-free implementation

is captured in Theorem 15 (Chap. 4), states that atomicity per se does not entail that a pending operation invocation has to wait for another operation invocation.

Wait-freedom is the strongest liveness condition that can be associated with mutex-free implementations. Given an object O, it states that any invocation of an operation on O issued by a non-faulty process has to terminate; i.e., the corresponding invocation has to terminate whatever the behavior of the other processes (which can be slow or even crashed). It is easy to see that only objects with total operations can be wait-free implemented.

Universal (synchronization) object and universal construction An object A of type T_a, or more generally a type T_a, is *universal* if any object Z whose type T_z is defined by a sequential specification on total operations can be wait-free implemented from (any number of) atomic read/write registers and objects of type T_a.

Any wait-free algorithm implementing such a construction is called a *universal construction*. The structure of a universal construction is depicted in Fig. 14.1.

14.1.2 The Notion of a Consensus Object

Binary consensus object was introduced in Sect. 6.3.1.We extend here the definition to multi-valued consensus. It will be shown in the rest of this chapter that consensus objects are universal objects.

Definition A consensus object provides a single operation denoted propose(v), where v is the value proposed by the invoking process. While only the values 0 and 1 can be proposed by processes to a binary consensus object, any value can be proposed to a multi-valued consensus object.

An invocation of propose() returns a value which is said to be the value *decided* by the invoking process. A process can invoke the operation propose() at most once (hence, a consensus object is a one-shot object). Moreover, any number of processes can invoke this operation. A process that invokes propose() is a *participating* process. The object is defined by the following properties. Let us recall that a process is correct in a run if it does not crash in that run; otherwise, it is faulty.

- Validity. A decided value is a proposed value.
- Integrity. A process decides at most once.
- Agreement. No two processes decide different values.
- Termination. An invocation of propose() by a correct process terminates.

The validity property relates the output to the inputs (a value that has not been proposed cannot be decided). The integrity property states that a decision is irrevocable. The agreement property defines the coordination power of a consensus object: no two processes can decide differently (in that sense, a consensus object solves non-determinism among the proposed values by selecting any but only one of them). Finally, the termination property states that the implementation has to be wait-free.

A consensus object is a one-write register Let $\perp$ denote a default value that cannot be proposed by a process. A consensus object C can be seen as maintaining an internal state variable X initialized to $\perp$. The effect of an invocation of propose(v) can be described by the atomic execution of the following code:

$$\textbf{if } (X = \perp) \textbf{ then } X \leftarrow v \textbf{ end if}; \textbf{ return } (X).$$

As we can see, a consensus object can be defined from a sequential specification. The operation propose() is a combination of a conditional write operation followed by a read operation. More generally, a consensus object can be seen as a one-write register that keeps forever the value proposed by the first invocation of propose(). Then, any subsequent invocation of propose() returns the value that was written. As a consensus object is a concurrent atomic object, if a process crashes while executing propose(), everything appears as if that invocation had been executed entirely or not at all.

14.2 Inputs and Base Principles of Universal Constructions

14.2.1 The Specification of the Constructed Object

Let Z be the object for which we want to build a wait-free implementation. This object is defined by a sequential specification that will be the input of a universal construction (see Fig. 14.1).

On the sequential specifications of the object Z The object Z is defined as a type that consists of a bounded set of m total operations $\mathsf{op}_1(param_1, res_1)$, …, $\mathsf{op}_m(param_m, res_m)$ and a sequential specification.

In the following, op($param$, res) is used to denote any of the previous operations. Each operation has a (possibly empty) set of input parameters ($param$) and returns a result (res). "Sequential specification" means that, given any initial state s_0 of Z, its behavior can be defined by the set of all the sequences of operation invocations where the output res of each invocation of an operation op() is entirely determined by the value of its input parameters $param$ and the operation invocations that precede it in the corresponding sequence.

Alternatively, the sequential specification can also be defined by associating a pre-assertion and a post-assertion with each operation. Assuming one operation at a time is executed on the object, the pre-assertion describes the state of the object before the operation while the post-assertion defines the result output by the operation and the new state of the object resulting from that operation execution. (Let us notice that, as the operations defining Z are total, all pre-assertions are always satisfied and are consequently equal to *true*.)

The sequential specification of Z used in the universal construction A sequence of operation invocations on an object can be abstracted as an object state, and accordingly, the semantics of each operation is defined by a transition function denoted $\delta()$.

More precisely, s being the current state of the object, $\delta(s, \texttt{oper}(param))$ returns a pair $\langle s', res\rangle$ from a finite non-empty set of pairs $\{\langle s[1]\, res[1]\rangle, \ldots, \langle s[x]\, res[x]\rangle\}$. Each pair of this set defines a possible output where s' is the new state of the object and res is the output value returned to the calling process.

If, for any $\delta(s, \texttt{oper}(param))$, the set $\{\langle s[1]\, res[1]\rangle, \ldots, \langle s[x]\, res[x]\rangle\}$ contains a single pair, the object is deterministic. Otherwise it is non-deterministic.

14.2.2 Base Principles of Universal Constructions

Internal representation of Z A universal construction is characterized by the way it represents the constructed object Z in the shared memory and the local memories of the processes. This internal representation is made up of a data part and a control part.

The first construction that is presented (Sect. 14.3) uses the shared memory only to support the control data used by the construction. Each process manages a copy of the data part representing Z. Differently, in the second construction that is presented (Sect. 14.4), both the control data and the value of Z are kept in the shared memory.

The role of consensus objects In both constructions, consensus objects are used by the processes to build a single total order in which the operation invocations are applied to Z. This is the method used to ensure that the internal representation of Z remains always consistent.

14.3 An Unbounded Wait-Free Universal Construction

The universal construction presented in this section is due to R. Guerraoui and M. Raynal (2007). It is based on the state machine replication paradigm. Each process maintains a copy of the object in its local memory, and the processes cooperate through consensus objects to keep their copies mutually consistent. Moreover, the processes use SWMR atomic registers to help each other in order to ensure the wait-freedom property.

This construction, which is relatively simple, is presented incrementally. It is first shown how consensus objects are used to order the operation invocations issued by the processes. It is then shown how atomic registers are used to ensure that no invocation of an operation issued by a correct process remains pending forever. As we will see, while the construction is wait-free, it is not bounded. This is because, when counting the number of operation invocations on the base objects that constitute the internal representation of Z, it is not possible to bound the period that elapses between the time an operation op() starts (event inv[op]) and the time a result is returned (event $resp$[op]).

The presentation concentrates first on the case of deterministic objects. It then addresses non-deterministic objects in Sect. 14.3.3.

14.3.1 Principles and Description of the Construction

Client versus server role The construction is an asynchronous algorithm in which each process p_i plays two roles, namely the role of a client when an operation is locally invoked and the role of a server when it cooperates with the other servers to wait-free implement the object Z. The first role is implemented by an appropriate sequence of statements executed each time an operation is locally invoked by p_i, while the second role is implemented by a set of background cooperating tasks (or threads), one for each process p_i. (Let us remember that, as each process is assumed to be sequential from the application layer point of view, a process can start a new invocation of an operation on Z only when the previous one has returned.)

The client behavior is described below (the line numbers correspond to the line numbers of the final construction described in Fig. 14.3). When Z.op($param$) is locally invoked, process p_i sets a local variable $result_i$ to $\perp$ (line 1) and informs its server thread that a new operation invocation was locally issued (line 2). To that end, another local variable denoted $prop_i$ is used. This variable is a pair containing the description of the operation "op($param$)" that was invoked and the identity i of the invoking process. Then, the client thread of p_i waits until the result associated with this invocation has been computed (line 3). When this occurs, the associated result is returned to the upper application layer (line 4). The threads that implement the server then have to cooperate so that the invocation of op($param$) issued by p_i be eventually executed in agreement with the sequential specification of Z.

```
operation $Z$.op($param$) is % locally invoked by $p_i$ %
(1) $result_i \leftarrow \perp$;
(2) $prop_i \leftarrow \langle$"op($param$)" , $i\rangle$;
(3) wait until ($result_i \neq \perp$);
(4) return($result_i$).
end operation.
```

Global structure: state machine replication As previously announced, this construction is based on the *state machine replication* paradigm. This approach consists in replicating a state machine on several servers and ensuring that all commands

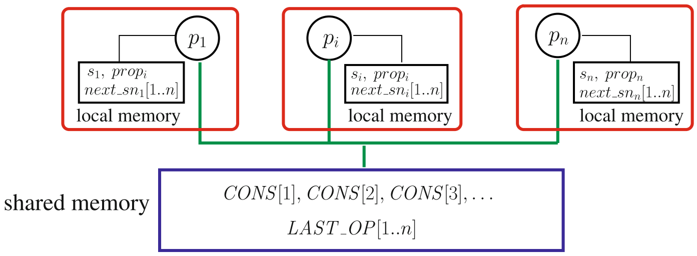


Fig. 14.2 Architecture of the universal construction

applied to that machine are applied in the same order on the copy managed by each server.

The concurrent object Z is here the state machine, and the operation invocations issued by the processes are the commands it has to execute. More precisely, we have the following:

- Each local server manages a local copy s_i of the constructed object Z. This copy s_i is initialized to the initial value of Z.
- In order that the servers apply the operation invocations in the same order to their local copies of Z, they cooperate through objects kept in the shared memory.

The global structure is described in Fig. 14.2.

Step 1 of the construction: use consensus objects to define a total order To apply the operation invocations in the same order to their local copies, the server threads are implemented by a background task at each process p_i. This task is the infinite loop described below (as before, the line numbers correspond to the line numbers of the final construction described in Fig. 14.3).

```
(5)  while ($true$) do
(12)     if ($prop_i \neq \bot$) then
(13)         $k_i \leftarrow k_i + 1$;
(14)         $exec_i \leftarrow CONS[k_i]$.propose($prop_i$);
(16)         $\langle s_i, res \rangle \leftarrow \delta(s_i, exec_i.op)$;
(17)         let $j = exec_i[r].proc$;
(19)         if ($i = j$) then $prop_i \leftarrow \bot$; $result_i \leftarrow res$ end if
(21)     end if
(22) end while.
```

When $prop_i \neq \bot$, the task discovers that a new operation invocation has locally been issued (line 12). So, in order both to inform the other processes and to guarantee that the operations will be applied in the same order to each copy, the local task proposes that operation invocation to its next consensus instance (lines 13–14). Hence, $\{CONS[k]\}_{k \geq 1}$, where $CONS[k]$, denote the kth consensus object instance.

As a value $prop_i$ proposed to a consensus instance is a pair, a decided value is also a pair, made up of an operation invocation with the identity of the invoking process (see line 2). So $exec_i$ (the local variable where the value decided by the last consensus instance is saved, line 14) is a pair composed of two fields, $exec_i.op$, which contains the decided invocation, and $exec_i.proc$, which contains the identity of the process that issued that invocation. The server executes then the state machine transition $\langle s_i, res\rangle \leftarrow \delta(s_i, exec_i.op\rangle$ to update its local copy s_i of Z (line 16). Moreover, if p_i is the process that has invoked this operation, it locally returns the result *res* by writing it into the local variable $result_i$ (line 19).

As (a) each consensus instance returns the same invocation to all the processes that invoke that instance and (b) the processes invoke the instances in the same order, it follows that, for any k, the processes p_i that have invoked the first k consensus instances applied the same length k sequence of operation invocations to their local copy s_i of the object Z.

If, after it has participated in k consensus instances (as just observed, s_i is then the state of Z resulting from the sequential application of the k operation invocations output by the first k consensus instances), a process p_i does not invoke an operation during some period of time, its local copy s_i is not modified. When, later, it invokes again an operation and proposes it to a consensus instance, it may have to execute consensus instances (starting from $k+1$) to catch up with a consensus instance where no value has yet been decided. While catching up, it will sequentially update its local state s_i according to the operation invocations that have been decided after the kth instance.

Non-blocking versus wait-free implementation Considering a consensus instance, exactly one process is a winner in the sense that the value proposed to that instance by that process is the decided value. In that sense the previous construction is *non-blocking*: all the processes that participate (i.e., propose a value) in a consensus instance and do not crash obtain the same value, and exactly one process (the winner) terminates its operation invocation.

Unfortunately, that implementation is not wait-free. Indeed, it is easy to build a scenario in which, while a process p_i continuously proposes the same operation invocation to successive consensus instances, it is never a winner because there are always processes proposing their operation invocations to these successive consensus instances and it is always a value proposed by one of these processes that is decided, the value proposed by p_i being never decided. In that case, the operation on Z invoked by p_i is never executed and, at the application level, p_i is prevented from progressing. It follows that, the construction is non-blocking but not wait-free.

Step 2 of the construction: introduce a helping mechanism A way to go from a non-blocking construction to a wait-free construction consists here in introducing a helping mechanism that allows a process to propose to a consensus instance not only its own pending invocation but all the pending invocations it is aware of. (Similar helping mechanisms have been used in previous chapters to build wait-free objects such as atomic registers or snapshot objects.)

To that end, an SWMR atomic register is associated with each process. That register allows its writer process to inform the other processes about the last operation it has invoked. More explicitly, *LAST_OP*[i] is an SWMR atomic register that can be written only by p_i and read by all the processes. When it writes its last operation invocation into $LAST\_OP[i]$, p_i "publishes" it, and all the other processes become aware of it when they read this register.

Such a register $LAST\_OP[i]$ is made up of two fields: $LAST\_OP[i].op$, which contains the last operation invoked by p_i with its input parameters (i.e., "op($param$)"), and $LAST\_OP[i].sn$, which contains a sequence number. $LAST\_OP[i].sn = x$ means that $LAST\_OP[i].op$ is the xth invocation issued by p_i. Each atomic register $LAST\_OP[i]$ is initialized to $\langle \perp, 0 \rangle$.

In order to know whether the last operation that p_j published in $LAST\_OP[j]$ has been or has not been applied to its local copy s_i of Z, each process p_i manages a local array of sequence numbers denoted $last\_sn_i[1..n]$ such that $last\_sn_i[j]$ contains the sequence number of the last operation invoked by p_j that was applied to s_i (for any j, $last\_sn_i[j]$ is initialized to 0).

Using the helping mechanism The previous helping mechanism is used as follows. When it invokes a consensus instance, a process p_i proposes all the operation invocations that have been published by the processes in $LAST\_OP[1..n]$ and that have not yet been applied to its local copy s_i of the object. From its point of view, those are all the invocations that have not yet been executed. This means that now, instead of its own invocation, a process p_i proposes a list of invocations to a consensus instance.

These design principles give rise to the universal construction described in Fig. 14.3. As the value $prop_i$ proposed by a process p_i to a consensus instance is a non-empty list of operation invocations, the value decided by that consensus instance is a non-empty list of invocations. Consequently, the local variable $exec_i$, where the value decided by the current consensus instance is saved, is now a list of invocations; $|exec_i|$ denotes its length and $exec_i[r]$, $1 \leq r \leq |exec_i|$, denotes its rth element. Let us recall that such an element is a pair (namely, the pair $\langle exec_i[r].op,\ exec_i[r].proc \rangle$).

14.3.2 Proof of the Construction

The proof of the universal construction described in Fig. 14.3 consists in showing that the constructed concurrent object Z is atomic and its operations are wait-free. Let us recall that atomicity means that, from an external observer point of view, everything has to appear as if there was a single copy of the object, the operations were executed one after the other on that copy and in an order that complies with the sequential specification of the object and respects their real-time occurrence order.

The proof is decomposed in several lemmas.

- A first lemma shows that the construction is wait-free; i.e., each operation invoked by a process that does not crash terminates despite the crash of any number of processes.
- A second lemma shows that any process sees the invocations issued by all the processes. Moreover, all the processes see them in the same total order. This allows one to show that all the local copies of the object Z are modified according

```
operation op($param$):
(1)  $result_i \leftarrow \bot$;
(2)  $LAST\_OP[i] \leftarrow \langle$“op($param$)”$, last\_sn_i[i] + 1\rangle$;
(3)  wait ($result_i \neq \bot$);
(4)  return($result_i$)
end operation.

Task $T$:    % background server task %
(5)  while ($true$) do
(6)      $prop_i \leftarrow \epsilon$;  % empty list %
(7)      for $j \in \{1, \dots, n\}$ do
(8)          if ($LAST\_OP[j].sn > last\_sn_i[j]$) then
(9)              append ($LAST\_OP[j].op, j$) to $prop_i$
(10)         end if;
(11)     end for;
(12)     if ($prop_i \neq \epsilon$) then
(13)         $k_i \leftarrow k_i + 1$;
(14)         $exec_i \leftarrow CONS[k_i]$.propose($prop_i$);
(15)         for $r = 1$ to $|exec_i|$ do
(16)             $\langle s_i, res\rangle \leftarrow \delta(s_i, exec_i[r].op)$;
(17)             let $j = exec_i[r].proc$;
(18)             $last\_sn_i[j] \leftarrow last\_sn_i[j] + 1$;
(19)             if ($i = j$) then $result_i \leftarrow res$ end_if
(20)         end for
(21)     end if
(22) end while.
```

Fig. 14.3 A wait-free universal construction (code for process p_i)

to the same sequence $\widehat{S}$ of operations. Everything appears then as if there was a single shared copy of Z and that copy satisfies its sequential specification. This is the place where both each consensus instance and the sequence of these instances come into play in order to establish a single order on all the operation invocations.

- A third lemma shows that $\widehat{S}$ respects the real-time occurrence order of all the operation invocations. This lemma defines the linearization points associated with each invocation of an operation.

Lemma 34 *The construction described in Fig. 14.3 is wait-free (i.e., each operation invocation issued by a correct process terminates).*

Proof Let us consider a correct process p_i (i.e., a process that does not crash) that invokes Z.op(*param*). It deposits the operation description with its sequence number into the shared register *LAST_OP*[i] to inform the processes on its pending operation (line 2). To show that a result is returned at line 4, we have to show that the predicate $result_i \neq \bot$ is eventually satisfied (line 3).

We claim (C) that (1) there is a consensus instance *CONS*[k] that outputs (at line 14) a list of pairs containing the pair ⟨ “op(*param*)”, i⟩ and (2) p_i participates in that consensus instance.

It follows from that claim that, when p_i executes the internal loop (lines 15–20) associated with the consensus instance $CONS[k]$, there is an internal loop iteration r during which $exec_i[r].proc = i$ and, consequently, p_i applies op($param$) to its local copy s_i of Z. During that loop iteration, the result of the operation is deposited in $result_i$, which proves the lemma.

Proof of the claim C. The proof is by contradiction. Let us assume that no consensus instance outputs a list containing the pair $\langle$ "op(param)", $i\rangle$.

Due to the test done to build a list proposal (line 8), it follows that there is a time after which all the lists proposed by the processes to consensus instances contain forever the pair $\langle$ "op(param)", $i\rangle$. As, from then on, $prop_i$ is never empty, process p_i participates in an infinite sequence of consensus instances with increasing sequence number k_i.

Let $CONS[k]$ be the first consensus instance such that each participating process proposes a list including the pair $\langle$ "op($param$)", $i\rangle$. As the invocations of $CONS[k]$.propose() issued by the correct processes that participate in that consensus instance return a list (consensus termination property) that is the same for all (consensus agreement property), and that list is the list proposed by one of them (consensus validity property), it follows that $exec_i$ contains the pair $\langle$ "op($param$)", $i\rangle$, which contradicts the initial assumption and proves the claim. *End of proof of the claim.* $\square$

Lemma 35 *All the operations invoked by the processes (except possibly the last operation invoked by process p_j, $1 \leq j \leq n$, if p_j crashes before depositing its operation invocation in $LAST\_OP[j]$) are totally ordered (this total order defines the sequence $\widehat{S}$).*

Proof We first show that any invocation Z.op() that is deposited by a process p_i in $LAST\_OP[i]$ is output by exactly one consensus instance.

Let us first observe that, as soon as p_i has deposited the pair $\langle$ "op($param$)", $i\rangle$ in $LAST\_OP[i]$ (line 2), it cannot invoke a new operation before this pair belongs to the list decided by a consensus instance (this follows from the management of the local variable $result_i$ at lines 1, 3, 17 and 19). It follows that, no operation invocation issued by a process can be overwritten before being output by a consensus instance.

The proof that the pair $\langle$"op($param$)", $i\rangle$ is eventually output by a consensus instance is the same as the proof of claim C in Lemma 34. We now have to prove that this operation is not output by more than one consensus instance. To that aim, assuming that several consensus instances can decide lists containing this pair, let k be the first of them (due to the previous discussion, there is at least one such instance). After it has obtained the list containing $\langle$"op($param$)", $i\rangle$ decided by the kth consensus instance, a process p_j increases $last\_sn_j[i]$ (line 18) before building a new proposal for the next consensus instance. Due to the test of line 8 used by p_j to build a new list and propose it to the next consensus instance, it follows that the pair $\langle$"op($param$)", $i\rangle$ can no longer be in the list proposed by p_j. We conclude then that, as soon as an operation invocation has been deposited in $LAST\_OP[i]$ by a process p_i, that invocation is decided by exactly one consensus instance.

The fact that the processes apply the same sequence of operations to their local copy of Z is a direct consequence of the use of consensus objects. Let us first observe that the processes use the consensus instances in the same order: first *CONS* [1], then *CONS* [2], etc. Moreover, each consensus instance orders a batch of invocations (the invocations that appear in the list decided by that instance). The combination of the single order on consensus instances and the single order on the invocations output by each consensus instance provides each correct process p_i with the same sequence $\widehat{S}$ of invocations of operation on Z. Moreover, as a process that crashes behaves as a correct process until it crashes, it follows that a process that crashes is provided with a prefix of the sequence $\widehat{S}$. □

It follows from the previous lemma and lines 12–21 of the universal construction that each correct process p_i applies to its local copy s_i of Z the same sequence of deterministic operations, and a process p_j that crashes applies to s_j a prefix of that sequence. We have consequently the following corollary:

Corollary 6 *The local copies s_i of all the correct processes behave as a single copy that complies with the sequential specification of Z.*

Lemma 36 *The sequence $\widehat{S}$ respects the real-time occurrence order on its operations.*

Proof Let op(*param*) be an invocation issued by a process p_i. Its start event *inv*[op] corresponds to the execution of line 1, and its response event *resp*[op] corresponds to the execution of line 4. If p_i crashes during this invocation, there is no response event.

Reminder. Let us remind that a linearization point associated with an operation invocation op(*param*) is a point of the time line that appears between the points associated with the events *inv*[op] and *resp*[op]. Its aim is to abstract the operation invocation and create the illusion that it had been executed instantaneously at that point of the time line. We have to show that it is possible to associate a linearization point with each invocation appearing in $\widehat{S}$ such that the total order on the linearization points is the total order defined by $\widehat{S}$. (End of reminder.)

Let $L_k \geq 1$ be the number of invocations decided and ordered by the consensus object $CONS[k]$, and op$[k, x]$ be the xth operation invocation ordered by $CONS[k]$. The sequence $\widehat{S}$ is then as follows: op$[1, 1], \ldots,$ op$[1, L_1]$, op$[2, 1], \ldots,$ op$[k, L_k]$, op$[k+1, 1], \ldots$ For each invocation, let us consider the consensus instance $CONS[k]$ and the first invocation of $CONS[k]$.propose() that returns the list of operation invocations op$[k, 1], \ldots,$ oper$[k, L_k]$. Let τ_k be the time at which this first invocation of $CONS[k]$.propose() starts. (This means that, if any, the other invocations of $CONS[k]$.propose() return later. This notion of "first/later" is well defined as the consensus objects are atomic. Moreover, p_i is not necessarily the process that has issued the first invocation of $CONS[k]$.propose().) We associate the linearization points (times) $\tau[k, 1], \ldots, \tau[k, L_k]$ with these invocations, such that $\tau_k < \tau[k, 1] < \cdots < \tau[k, L_k] < \tau_{k+1}$. Let us observe that the total order on these linearization points is the same as the total order defined by $\widehat{S}$.

It remains to show that the linearization point of each invocation $\mathsf{op}(param)$ is between its associated events $inv[\mathsf{op}]$ and $resp[\mathsf{op}]$ (if it exists). Let us assume that $\mathsf{op}(param)$ was issued by p_i and the associated linearization point is $\tau[k, x]$. Let us first observe that, for the pair $\langle$“$\mathsf{op}(param)$”, $i\rangle$ to be decided by a consensus instance, it has first to be deposited in $LAST\_OP[i]$, from which it follows that $\tau[k, x]$ is necessarily after $inv[\mathsf{op}()]$. For the event $resp[\mathsf{op}]$ to occur, p_i must invoke $CONS[k].\mathsf{propose}()$ and obtain the list of invocations decided by that consensus instance. If this list contains $\langle$“$\mathsf{op}(param)$”, $i\rangle$, it follows that p_i first executes line 16 ($\mathsf{op}(param)$ is applied to s_i) and only then the event $resp[\mathsf{op}]$ is allowed to occur. It follows that, if $resp[\mathsf{op}]$ occurs, it occurs after $\tau[k, x]$, which proves the lemma. □

Theorem 57 *Let Z be a concurrent object defined by a sequential specification on total deterministic operations. The construction described in Fig. 14.3 is a wait-free implementation of the atomic object Z.*

Proof The wait-free property is proved in Lemma 34. Lemma 35 has shown that $\widehat{S}$ is a sequential history including all invocations. Corollary 6 has shown that $\widehat{S}$ is legal (i.e., respects the sequential specification of the object Z). Finally, Lemma 36 has shown that $\widehat{S}$ respects the real-time order on the operation invocations. It follows that the construction builds a wait-free atomic object. □

14.3.3 Non-deterministic Objects

When the object Z is non-deterministic, the function $\delta()$ can return any pair from a set including more than one pair, and consequently two different processes p_i and p_j can obtain different pairs for the same state transition, thereby entailing divergence of their local copies of Z. When this occurs, the construction algorithm described in Fig. 14.3 no longer ensures that the all the local copies of Z behave as a single copy. This section describes three solutions that allow the implementation of non-deterministic objects Z.

A brute force solution A brute force strategy to solve the previous inconsistency problem consists in replacing the non-deterministic transition function $\delta()$ by any one of its deterministic restrictions $\delta'()$. More precisely, the transition function $\delta'()$ is restricted to be such that, for any state s and any operation invocation $\mathsf{op}(param)$, $\delta'\big(s, \mathsf{op}(param)\big)$ has a single possible output.

Using additional consensus objects to cope with non-determinism A solution preserving the non-deterministic dimension of Z consists in using additional consensus objects $\{CONS\_ND[k]\}_{k\geq 1}$ to prevent possible divergences. Each time the transition function $\delta\big(s_i, \mathsf{op}(param)\big)$ can return a pair $\langle s, res\rangle$ from a set including at least two pairs (line 16), each process p_i proposes to the instance $CONS\_ND[k]$ the pair $\langle s, res\rangle$ it has obtained from $\delta\big(s_i, \mathsf{op}(param)\big)$. To that end, p_i invokes

$CONS\_ND[k]$.propose($\langle s, res \rangle$). As the same pair is deterministically imposed on all the processes that invoke $CONS\_ND[k]$.propose(), it follows that they all obtain the same pair $\langle s', res' \rangle$ (proposed by one of them). Each of them has then to execute the statements $s_i \leftarrow s'$ and $res \leftarrow res'$.

Solving non-determinism without additional consensus objects It is possible to devise a simple solution that does not use additional consensus objects. The idea is to use the consensus objects $\{CONS[k]\}_{k \geq 1}$ in a preventive manner to agree (1) on the ordering of the operations inside a proposed list of operation invocations and the result of each of these invocations (as before) and (2) on the new state of the copy s_i of the object.

More specifically, before invoking $CONS[k]$.propose(), starting from its local state s_i of Z, a process p_i executes the invocations in the list $prop_i$ and computes the result associated with each of these invocations and the resulting state of Z. It then proposes to the consensus instance $CONS[k]$ a triple made up of the list $prop_i$, the corresponding list of results, and the corresponding state of Z. According to the triple decided by $CONS[k]$, a process executes consequently the lines 18–19 for each invocation of the decided list, and then installs the new state of its local copy of Z.

The construction obtained in this way solves non-determinism without incurring additional cost in terms of consensus objects and shared registers. The only additional cost is in the additional processing and in the size of the values proposed to consensus objects.

14.3.4 Wait-Freedom Versus Bounded Wait-Freedom

Finite versus bounded Lemma 34 has shown that the previous construction is wait-free (each operation issued by a correct process terminates). We now show that it is *not bounded wait-free*. This means that it is not possible to state a bound on the number of operation invocations on base objects (consensus object and atomic read/write registers) that need to be executed before an invocation of an operation Z.op() terminates. This number is finite (hence the wait-free property), but there is no bound that would hold in any execution.

Why the construction of Fig. 14.3 is not bounded wait-free Considering the construction described in Fig. 14.3, let p_i be a process that invokes an operation op(). Moreover, let k_{inv} be the value of k_i when p_i invokes op() and $CONS[k_{resp}]$ (where $k_{inv} < k_{resp}$) be the consensus instance that outputs op(); i.e., op() belongs to the list decided by $CONS[k_{resp}]$. This means that the task T of p_i has to execute $K = k_{resp} - k_{inv}$ times the lines 6–21 in order to catch up with the consensus instance in which its invocation op() is decided. The proof of Lemma 34 has shown that K is finite. We show that it cannot be bounded.

To show this, consider the case where no operation is invoked by p_i and its task T is "sleeping" during an arbitrary long period (this is possible, due to asynchrony). During that period, the other processes p_j issue an arbitrarily large number of operation invocations and their values k_j can become arbitrarily large. Then, at some time τ after that arbitrary long period, the process p_i invokes an operation and its task T wakes up (let us observe that, despite the fact that the task T of a process is not synchronized with its own operation invocations, this is a possible scenario). The value K is then arbitrarily large, which shows that it cannot be bounded. Hence, the construction is not bounded wait-free.

14.4 A Bounded Wait-Free Universal Construction

This section presents a bounded wait-free construction of an atomic object Z from consensus objects and MWMR atomic registers. This construction, which is due to M. Herlihy (1991), is the first universal construction that was proposed. As in the other universal constructions, consensus objects are used to totally order the operation invocations. The atomic registers are used to obtain the bounded wait-freedom property. Differently from the previous construction, where all the atomic registers were SWMR, some atomic registers are now SWMR while others are MWMR. As before, we first consider the case of deterministic objects, and then non-deterministic objects.

14.4.1 Principles of the Construction

As we have seen in Sect. 14.3.4, what prevents the construction described in Fig. 14.3 from being be bounded is the fact that each process has its own copy of Z and does not strive to keep its local copy up to date when it does not invoke operations on Z.

The ideas on which Herlihy's bounded wait-free construction relies are (1) a single copy of the object Z (maintained in shared memory) plus (2) a helping mechanism that allows bounding of the number of invocations of base object operations that can be executed between the invocation of an operation and the return of the corresponding result.

Internal representation of the object The object is represented as a linked list, where the sequence of cells represents the sequence of operation invocations applied to the object. A process executes an operation by adding a new cell to the list. So, there is a single (centralized) representation of the object.

A cell is dynamically created by a process p_i each time it invokes an operation. It is made up of five fields (Fig. 14.4):

- The field *sn* is a sequence number initialized to 0 when the cell is created. Then, it takes a positive value (its rank in the list) and keeps it forever.
- The field *invoc* contains the description of the operation (with its parameters "op(*param*)") invoked by the process p_i that created the cell.
- The fields *state* and *res* contain a pair of related values:

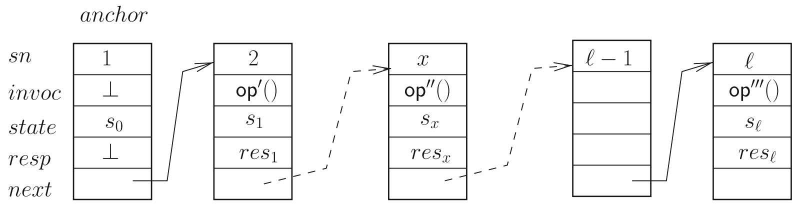


Fig. 14.4 The object Z implemented as a linked list

- *state* contains the state of the object Z after sequential application of all the operations in the list, from the first up to and including the operation op(*param*).
- *resp* is the result associated with the invocation op(*param*).

From an operational point of view, these fields are computed from the state of the object defined in the immediately preceding cell of the list. Let *prev_state* be this state. Given a cell, we have $\langle state, resp\rangle = \delta\big(prev\_state, \mathsf{op}(param)\big)$.

- The last field, denoted *next*, is a consensus object created together with the cell. The aim of this consensus object is to contain the pointer to the next cell of the list. As a consensus object decides a single value, any cell can point towards a single next cell. This is the way the consensus objects are used to create a total order on the operation invocations issued by the processes.

A cell is not an atomic object, while each of its components is atomic (four atomic registers plus a consensus object). While the field *invoc* refers to an SWMR register (only the process that creates a cell can write this field), the three other fields *sn*, *state*, and *resp* are MWMR atomic registers.

Initially, the list is represented by a unique cell, called *anchor*. Its field *sn* has the value 1, while its field *state* has the value s_0 (the initial state of the object). Its fields *invoc* and *resp* are irrelevant. As far as the field *next* is concerned, let us recall that a consensus object has no initial value.

Ensuring the bounded wait-freedom property To ensure the bounded wait-free property, each process p_i manages two atomic SWMR registers, denoted *LAST_OP*[i] and *HEAD*[i], the values of which are pointers to cells. Initially, both *LAST_OP*[i] and *HEAD*[i] points to the cell *anchor*. As far as terminology is concerned, the last cell added to the list defines its "head". More specifically, we have the following. (The line numbers refer to Fig. 14.5.)

- The role of the register *LAST_OP*[i] is similar to that of the previous construction, namely, it is used by p_i to make public its last invocation of an operation on Z so that the other processes can help it if needed. Hence, *LAST_OP*[i] is used to ensure the wait-freedom property.

When p_i invokes an operation op(*param*), it creates a new cell denoted *CELL*, initializes its *sn* field to 0 and its *invoc* field to the description of the operation

```
operation op($param$) is
(1)  allocate a new cell $CELL$;
(2)  $CELL \leftarrow$(0, “op($param$)”, $\perp$, $\perp$, new consensus object);
(3)  $LAST\_OP[i] \leftarrow (\uparrow CELL)$;
(4)  $HEAD[i] \leftarrow HEAD[x]$ such that $(HEAD[x] \downarrow).sn = \max\{HEAD[y] \downarrow).sn\}_{1 \le y \le n}\}$;
(5)  $last\_sn_i \leftarrow (HEAD[i] \downarrow).sn$;
(6)  while $((LAST\_OP[i] \downarrow).sn = 0)$ do
(7)        $to\_help \leftarrow (last\_sn_i \mod n) + 1$;
(8)        if $(LAST\_OP[to\_help] \downarrow).sn) = 0$ then $prop_i \leftarrow LAST\_OP[to\_help]$
(9)                                                 else $prop_i \leftarrow LAST\_OP[i]$   end if;
(10)       let $CONS = (HEAD[i] \downarrow).next$;
(11)       $next\_cell \leftarrow CONS$.propose($prop_i$);
(12)       $(next\_cell \downarrow).\langle state, res\rangle \leftarrow \delta\,((HEAD[i] \downarrow).state,\ (next\_cell \downarrow).invoc)$;
(13)       $last\_sn_i \leftarrow last\_sn_i + 1$; $(next\_cell \downarrow).sn \leftarrow last\_sn_i$;
(14)       $HEAD[i] \leftarrow next\_cell$
(15) end while;
(16) $result_i \leftarrow (LAST\_OP[i] \downarrow).res$;
(17) return($result_i$)
end operation.
```

Fig. 14.5 Herlihy’s bounded wait-free universal construction

invocation ”op(*param*)”, and then makes *LAST*_*OP*[*i*] to point to that cell (lines 1–3 in Fig. 14.5). After this has been done, the invocation op(*param*) issued by p_i is published and can consequently be executed by any process.

Then, $(LAST\_OP[i] \downarrow).sn \neq 0$ indicates that the cell associated with the last invocation issued by p_i has been added to the linked list; i.e., the invocation saved in $(LAST\_OP[i] \downarrow).invoc$ was executed (lines 6 and 13).

- The atomic register *HEAD*[*i*] is used to ensure the boundedness attribute of the wait-freedom property. This atomic register contains a pointer to the last cell of the linked list as known by p_i. This means that $(HEAD[i] \downarrow).state$ is the current state of the constructed object Z known by p_i. Let us notice that, due to asynchrony, two distinct processes p_i and p_j do not necessarily have the same view of which is the last cell added to the list; i.e., *HEAD*[*i*] can be different from *HEAD*[*j*], and both can be different from the cell which is currently the head of the list.

Description of the universal construction Herlihy’s bounded wait-free construction is described in Fig. 14.5. Let us first observe that a process that invokes no operation does not participate in the algorithm. It follows that the construction is *adaptive*: a process is required to help other processes only when it invokes an operation.

When an operation is locally invoked, p_i executes two computation phases before returning the result associated with the operation it has invoked (lines 16–17).

- The first phase is a preparation and announcement phase (lines 1–5).
  First, p_i associates a new cell to its invocation (as described above in the presentation of *LAST*_*OP*[*i*], lines 1–5) and asynchronously computes a local view

of which is the last operation invocation that was added to the list. This view, computed from the sequence numbers associated with the cells pointed to by $HEAD[1..n]$, is saved in $HEAD[i]$ (line 4).

The local variable $last\_sn_i$ (line 5) represents then, from p_i's point of view, the sequence number associated with the last operation invocation that was executed (i.e., added to the linked list). It is important to see here that the processes help each other: a process reads the last sequence number known by each of them, namely it reads all the registers $(HEAD[1..n] \downarrow).sn$ (line 4), in order to obtain the best view of which is the last operation invocation that was executed.

Remark Let us notice that a process p_i updates $LAST\_OP[i]$ (line 3) before reading asynchronously the entries of the array $HEAD[1..n]$ (line 4). The fact that $LAST\_OP[i]$ is updated first is important to ensure the boundedness attribute of the wait-free property, as we will see in the proof of Lemma 37. End of remark.

- The second phase is a *helping* and computation phase (lines 6–15).

  Its aim is to determine the result associated with the operation invoked by p_i which occurs when the corresponding cell is added to the list.
  First, the helping mechanism is used. Inspired from the classical round-robin principle, this mechanism works as follows. The priority to add the kth cell to the list is systematically given to a process p_x if

  – That process has a pending operation (i.e., $(LAST\_OP[x] \downarrow).sn = 0$), and

  – Its index x is such that $x = (k \mod n) + 1$.

  This is expressed at lines 7–8 (where $k = last\_sn_i$ and $x = to\_help$). It is easy to see that, for each sequence number, exactly one process is given priority and no process that has a pending operation can be missed.

  Combined with the management of the array $HEAD[1..n]$, this ensures that, as soon as a process has made public its operation (line 3), at most one operation invoked by each other process can be added to the list before its invocation. Hence, the bounded wait-free property. This is operationally expressed in the construction at lines 8–9, where $prop_i$ is a pointer to the cell that p_i has to try to add to the list.

  The addition of a cell into the list is as follows. p_i considers the cell that (from its point of view) is at the head of the list (namely, the cell pointed to by $HEAD[i]$) and tries to append after it the cell it has previously determined, that is, the cell pointed to by $prop_i$ (lines 10–12). This is where the consensus objects come into play. As already mentioned, the field *next* of a cell is a consensus object destined to contain the pointer to the next cell. Here the relevant consensus object to thread the next cell is $(HEAD[i] \downarrow).next$. Let us recall that, if several processes try to thread different cells, a single one will succeed (this is because $(HEAD[i] \downarrow).next$ is a consensus object, and *next_cell* then contains the pointer value decided by that consensus object (line 11).

Then, p_i executes the operation encapsulated in *next_cell* (line 12). As the operations are deterministic, if several process write into the pair of atomic registers $(next\_cell \downarrow).\langle state, res\rangle$, they write the same pair of values. The field $(next\_cell \downarrow).sn$ is then updated to $(HEAD[i] \downarrow).sn + 1$ (line 13), and $HEAD[i]$ is advanced to *next_cell* at line 14.

Finally, the process p_i terminates looping when its invocation was threaded into the list. This is operationally detected when the predicate $(LAST\_OP[i] \Downarrow).sn \neq 0)$ becomes true. Let us observe that this predicate can be satisfied when p_i evaluates it for the first time at line 6 (this occurs when p_i is slow: after it has announced its operation invocation in $LAST\_OP[i]$, that invocation has been threaded into the list by another process before p_i starts executing line 7).

14.4.2 Proof of the Construction

Lemma 37 *Herlihy's construction (Fig. 14.5) is bounded wait-free.*

Proof Let us first remember that the base objects are the n atomic registers $HEAD[1..n]$, the n atomic registers $LAST\_OP[1..n]$ plus four atomic registers and a consensus object per cell.

Let p_i be a process that invokes an operation $Z.\mathsf{op}(param)$. If p_i finds $(LAST\_OP[i] \downarrow).sn \neq 0$ when it checks for the first time the predicate of line 6, its operation invocation has already been threaded into the list and it terminates in a constant number of its own operations on base objects (namely, the operations it has executed at lines 1–6, and lines 6, and 17). The construction is consequently bounded wait-free for this case. So, the rest of the proof considers the case where p_i finds $(LAST\_OP[i] \downarrow).sn = 0$ when it checks for the first time the predicate of line 6.

As the base objects are atomic, any execution considered at the level of the base object operations is linearizable (Theorem 14, Chap. 4). This observation allows us to reason on the total order (on the base object operations) defined by such a linearization. So, we use the global time frame defined by such a linearization. In the following, $X(\tau)$ denotes the value of the base object X at time τ. Let us first make the following two observations that are immediate consequences of the code of the construction (these observations are implicitly used in the proof):

- Observation O1: $\forall i: \forall \tau, \tau': (\tau \leq \tau') \Rightarrow \big((HEAD[i] \downarrow).sn(\tau)\big) \leq \big((HEAD[i] \downarrow).sn(\tau')\big)$.
- Observation O2: Let us consider a process p_i. Each time p_i executes the body of the **while** loop (lines 6–15) it locally observes that (a) one more cell is threaded into the list (due to the invocation $CONS.\mathsf{propose}(prop_i)$ at line 11) and (b) $(HEAD[i] \downarrow).sn$ is increased by 1 (lines 13–14).

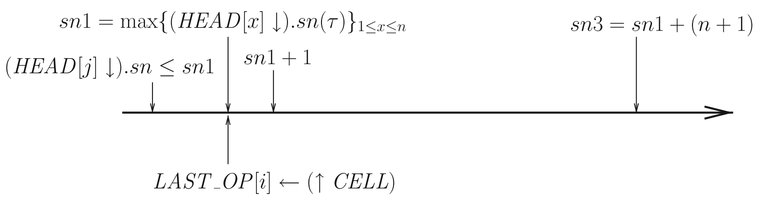

$sn1 = \max\{(HEAD[x] \downarrow).sn(\tau)\}_{1 \le x \le n}$
$sn3 = sn1 + (n+1)$
$(HEAD[j] \downarrow).sn \le sn1$
$sn1 + 1$
$LAST\_OP[i] \leftarrow (\uparrow CELL)$

Fig. 14.6 Sequence numbers

Let τ be the time at which p_i has executed line 3 (i.e., just after $LAST\_OP[i]$ has been updated). Let $sn1$ be the value of $\max\{(HEAD[x] \downarrow).sn\}_{1 \le x \le n}$ at time τ. Let $sn3 = sn1 + (n + 1)$ (see Fig. 14.6).

We claim that $LAST\_OP[i]$ is threaded into the list with a sequence number smaller than or equal to $sn3$. It follows from this claim that p_i executes at most $n + 1$ times the body of the **while** loop. As each loop iteration is made up of a bounded number of operations on base objects, it follows that p_i executes a bounded number of invocations on base objects (each base operation—be it a read or a write of a shared register, or a propose() invocation on a consensus object—counts for 1 when counting the number of base object operations executed by p_i). Hence, there are $O(n)$ invocations on base objects between the time instant at which p_i invokes op(*param*) and the time instant at which p_i executes the return() statement (line 17). The construction is consequently bounded wait-free.

Proof of the claim. The proof is by contradiction. Let us assume that $LAST\_OP[i]$ is not threaded into the list with a sequence number smaller than or equal to $sn3$. From τ, and until the cell pointed to by $LAST\_OP[i]$ is threaded into the list, all the processes that read $(LAST\_OP[i] \downarrow).sn$ obtain the value 0 (line 8). As $sn3 = sn1 + (n + 1)$, there is at least one sequence number $sn2$ (and at most two) such that $i = (sn2 \mod n) + 1$ and $sn1 < sn2 \le sn3$. When there are two such sequence numbers, those are $sn1 + 1$ and $sn3 = sn1 + (n + 1)$. We consider two cases:

- There is a single sequence number $sn2$ such that $i = (sn2 \mod n) + 1$ and $sn1 + 1 \le sn2 \le sn3$. In this case, $sn1 + 1 < sn2 < sn3 + 1$.
  As $sn2 > sn1+1$, one or several cells have been threaded into the list with sequence numbers $sn1+1,\dots, sn2-1$ when, after τ, a process reads $last\_sn_j = sn2-1 > sn1$ (at line 5 or 13). It follows that, when this occurs, such a process necessarily obtains 0 when thereafter it reads the atomic register $(LAST\_OP[i] \downarrow).sn$ (line 13). It follows from this reading that such a process p_j sets $prop_j$ to $LAST\_OP[i]$ (line 8). This is true for all the processes that execute the loop when their $last\_sn$ local variable becomes equal to $sn2 - 1$. Hence, all the processes that try to add the next cell (i.e., the $sn2$th cell) to the list propose $LAST\_OP[i]$ to the consensus object of the $(sn2 - 1)$th cell. Moreover, let us notice that at least one process executes the loop with $last\_sn_j = sn2 - 1$ (namely p_i, as, due to the contradiction assumption, its operation is not added before the sequence number $sn3+1$ and we consequently have $(LAST\_OP[i] \downarrow).sn = 0$). It follows that at least one process invokes the base

operation propose($LAST\_OP[i]$) on the consensus object of the $(sn2-1)$th cell. It follows from the validity property of that base consensus object that the $sn2$th cell that is threaded is $LAST\_OP[i]$, which contradicts the initial assumption and proves the claim.

- There are two sequence numbers $sn2$ such that $i = (sn2 \mod n)+1$ and $sn1+1 \leq sn2 \leq sn3$. In that case, these numbers are $sn1+1$ and $sn3$.
  It is possible that some processes p_j obtain $last\_sn_j = sn1$ before τ (see Fig. 14.6). If this happens, these processes can thread a cell different from the one pointed to by $LAST\_OP[i]$ with sequence number $sn2 = sn1 + 1$. This is because these processes try to add the $sn2$th cell before reading 0 from $(LAST\_OP[i] \downarrow).sn$, and although this sequence number gives priority to the cell announced by p_i, those processes are not yet aware of that cell. So, $LAST\_OP[i]$ misses its turn to be threaded. But, after this addition, we are after τ and all the processes see $(LAST\_OP[i] \downarrow).sn = 0$ when the priority is again given to p_i. It follows from the reasoning of the previous case that the cell pointed to by $LAST\_OP[i]$ will be threaded at the latest as the $sn3$th cell into the list. *End of proof of the claim.* □

Lemma 38 *All the operation invocations (except the last invocations of the processes p_j that crash before depositing their invocation in $LAST\_OP[j]$) are totally ordered. Let $\widehat{S}$ be that sequence of invocations. $\widehat{S}$ belongs to the sequential specification of Z.*

Proof Each operation invocation issued by a process (except possibly its last operation if it crashes) is associated with a single cell that (as shown by Lemma 37) is threaded into the list with the help of a consensus object. Due to the very definition of a list (a list is a sequence), it follows that all the operations define a sequence $\widehat{S}$.

Moreover, when considering the list of cells, this sequence is defined starting from the initial state s_0 of the object Z (as defined in the cell *anchor*), and then going from one cell to the next one by applying the transition function $\delta()$ defined by the sequential specification of Z (line 12). It follows that $\widehat{S}$ belongs to the sequential specification of the constructed object Z. □

Lemma 39 *The sequence $\widehat{S}$ respects the real-time occurrence order on its operation invocations.*

Proof Let us define the linearization point of an operation invocation Z.op(*param*) as the time when the corresponding cell (built at line 2) is added to the list (line 11).

This means that the linearization point corresponds to the first invocation of propose() on the consensus object *CONS* that threaded op(*param*) into the list. This point in time is trivially between the time the operation was invoked (when we had $(LAST\_OP[i] \downarrow).sn = 0$) and the time a result is returned (when we had $(LAST\_OP[i] \downarrow).sn \neq 0$). Moreover, as each consensus instance outputs a single pointer to a cell and the consensus instances define the total order $\widehat{S}$ on the operation invocations, it follows that the total order on the linearization points is the same as the total order $\widehat{S}$ on the corresponding operation invocations, which proves the lemma. □

Theorem 58 *Herlihy's construction (Fig. 14.5) is a bounded wait-free construction of any concurrent object Z defined by a sequential specification on total deterministic operations.*

Proof The theorem follows from Lemma 37 (bounded wait-freedom property), Lemma 38 ($\widehat{S}$ is a sequential history that includes all the operation invocations and belongs to the sequential specification of Z), and Lemma 39 ($\widehat{S}$ respects the real-time order on the operation invocations). $\square$

14.4.3 Non-deterministic Objects

This section considers the case where operations on Z are not deterministic. When considering the previous construction, the line where a non-deterministic operation can create a problem is line 12, namely, the line where the pair $\langle state, res\rangle$ of a cell is written according to the values output by $\delta\big(prev\_state, \mathsf{op}(param)\big)$.

As these fields of a cell can be written by several processes, different processes can now write different pairs of values. As between any two such writings these values can be read by other processes, the linked list representing the object Z will no longer satisfies its sequential specification.

A simple way to address this problem consists in demanding that the processes reach consensus on the pair $\langle state, res\rangle$ of each cell. In that way, such a pair is made unique for each cell. At the operational level, this can be done as follows:

- In addition to its previous fields, a cell has now a new field denoted *state_res_cons* which contains a consensus object. This consensus object of the cell *anchor* is assumed to be properly initialized to $\langle s_0, \perp\rangle$ (this can easily be done by requiring each process to invoke the operation instance *anchor.state_res_cons*. $\mathsf{propose}(\langle s_0, \perp\rangle)$ when it starts).
- Line 12 of Fig. 14.5 is replaced by the following lines (where $prop\_pair_i$ is an auxiliary local variable of p_i that can contain a pair of values):

```
(12.a)  $prop\_pair_i \leftarrow \delta\big((HEAD[i]\downarrow).state,\ (next\_cell\downarrow).invoc)\big)$;
(12.b)  let $CONS\_ND = (HEAD[i]\downarrow).state\_res\_cons$;
(12.c)  $(next\_cell\downarrow).\langle state, res\rangle \leftarrow CONS\_ND.\mathsf{propose}(prop\_pair_i)$.
```

14.5 From Binary Consensus to Multi-Valued Consensus

As we have previously seen in the book, a binary consensus object is a consensus object to which only two values (usually denoted 0 and 1) can be proposed, while there is no such restriction for multi-valued consensus objects.

This section shows that binary consensus is not less powerful than multi-valued consensus. To that end, it presents two constructions that build a multi-valued consensus object on top of an asynchronous system prone to any number of process crashes, and where processes communicate through atomic read/write registers and binary consensus objects. The first construction assumes that the domain of values that can be proposed is bounded and the bound is known by the processes, while the second construction does not restrict the domain of values that can be proposed.

The constructed multi-valued consensus object is denoted $CONS$. To prevent confusion, the consensus operation propose() is denoted mv_propose() for the constructed multi-valued consensus object and bin_propose() for the underlying binary consensus objects.

14.5.1 A Construction Based on the Bit Representation of Proposed Values

Preliminary notations The arrays that are used are arrays of b bits, where b is the number of bits required to encode any value that can be proposed. Hence, the set of values that can be proposed is bounded. It is $\{0, 1, \ldots, 2^b - 1\}$. It is assumed that b is known by the processes.

Let $aa[1..b]$ be such an array and $0 \leq k < b$. The notation $aa[1..0]$ denotes an empty array (it has no entries), and $aa[1..k]$ denotes the sub-array containing the entries from 1 to k. Finally, the predicate $aa[1..k] = bb[1..k]$ is true if and only if these two sub-arrays of bits are component-wise equal. By definition, the predicate $aa[1..0] = bb[1..0]$ is always satisfied.

Internal representation of the multi-valued consensus object The internal representation of the multi-valued consensus object $CONS$ is made up of two arrays of n elements (where n is the number of processes):

- $PROP[1..n]$ is an array of SWMR atomic registers initialized to $[\perp, \ldots, \perp]$.

  Then, the aim of each $PROP[i]$ is to contain the b-bit array-based representation of the value proposed by the process p_i. Hence, when $PROP[i] \neq \perp$, $PROP[i][1..b]$ contains the value proposed by p_i and its kth bit is kept in $PROP[i][k]$.

- $BC[1..b]$ is an array of b binary consensus objects.

Principle of the algorithm The principle of the algorithm is simple: the processes agree sequentially, one bit after the other, on each of the b bits of the decided value. This is done with the help of the underlying binary consensus objects $BC[1..b]$.

In order for the decided array of bits to correspond to one of the proposed values, when the processes progress from bit k to bit $k + 1$, they consider only the binary coding of the proposed values whose first k bits are a prefix of the sequence of bits on which they have already agreed. A process p_i keeps this sequence of agreed bits in a local array $res_i[1..b]$ of which only the sub-part $res_i[1..k]$ is meaningful.

```
operation $CONS$.mv_propose $(v_i)$ is
(1)  $PROP[i] \leftarrow b$-bit array-based binary representation of $v_i$; $k \leftarrow 0$;
(2)  repeat
(3)      let $compet\_set = \{x \mid PROP[x] \neq \bot) \wedge (PROP[x][1..k] = res_i[1..k])\}$;
(4)      $k \leftarrow k + 1$;   % next bit to agree on %
(5)      let $bit\_set = \{PROP[x][k] \mid x \in compet\_set\}$;   % $bit\_set = \{0\}$ or $\{1\}$ or $\{0,1\}$ %
(6)      let $bp_i$ = any bit $\in bit\_set$;
(7)      $res_i[k] \leftarrow BC[k]$.bin_propose$(bp_i)$;
(8)  until $(k = b)$ end repeat;
(9)  return(the value encoded by $res_i[1..b]$)
end operation.
```

Fig. 14.7 Multi-valued consensus from binary consensus: construction 1

Behavior of a process The construction is described in Fig. 14.7. A process p_i first deposits in $PROP[i]$ the binary coding of the value v_i it proposes (line 1). Then, p_i enters a loop that it executes b times (one loop iteration per bit). The local integer k, $0 \leq k \leq b$, is used to denote the number of bits on which the processes have already agreed. As already indicated, the value of these k bits is locally saved in the local sub-array $res_i[1..k]$. The loop is as follows:

- A process p_i first computes the set (denoted *compet_set*) including the processes p_x that have proposed a value the first k bits of which are the same as the k bits on which p_i has already agreed, namely the sub-array $res_i[1..k_i]$ (line 3).
- Then, p_i proceeds to the computation of the next bit of the decided value (statement $k \leftarrow k+1$, line 4). It computes the set of bits (*bit_set*) that are candidates to become the next bit of the decided value (line 5) and proposes one of these bits to the next binary consensus instance $BC[k]$ (lines 6–7).

  Finally, the value decided by $BC[k]$ is saved in $res_i[k]$ (line 7). After this statement, there is an agreement on the k first bits of the decided value. If $k = b$, the decided value was computed and p_i returns it. Otherwise, it proceeds to the next loop iteration.

Theorem 59 *The construction described in Fig. 14.7 is a wait-free implementation of a multi-valued consensus object from read/write atomic registers and binary consensus objects.*

Proof (Sketch) The proof is tedious, so we give only a sketch of it. The consensus termination property is a consequence of the following observations: (a) the number of iterations executed by a process is bounded, (b) the sets *compet_set* and *bit_set* computed at line 3 and line 5 are never empty, and (c) for any k, any correct process that invokes $BC[k]$.bin_propose(bp_i) terminates.

The agreement property follows from the fact that, when a process p_i proposes a bit bp_i to the binary consensus object $BC[k]$, all the processes that have executed $BC[j]$.bin_propose() for $1 \leq j < k$ agree on the same prefix of $(k-1)$ bits of the decided value.

Finally, the consensus validity property (a decided value is a proposed value) follows from the fact that, at the beginning of each loop iteration k, $0 \leq k < b$, the proposed values that are known by p_i (the non-$\bot$ values in $PROP[1..n]$) and whose prefix $PROP[x][1..k]$ does not match the prefix $res_i[1..k]$ that has already been computed are discarded by p_i before it selects the next bit it will propose to the binary consensus $BC[k + 1]$. □

14.5.2 A Construction for Unbounded Proposed Values

Differently from the previous construction, this one places no constraint on the size of the values that can be proposed.

Internal representation of the multi-valued consensus object This internal representation is very similar to the one of the previous construction:

- $PROP[1..n]$ is an array of SWMR atomic registers initialized to $[\bot, \ldots, \bot]$. The aim of $PROP[i]$ is to contain the value proposed by process p_i.
- $BC[1..n]$ is an array of n binary consensus objects.

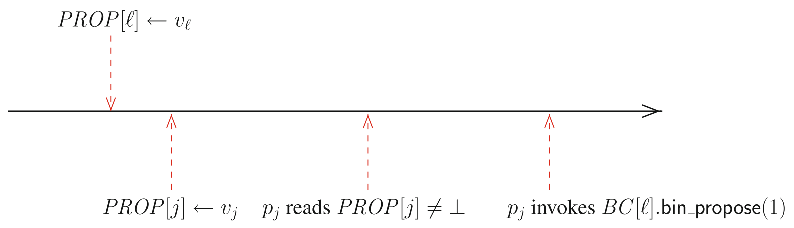
Principles of the construction The construction is described in Fig. 14.8. A process p_i first deposits the value it proposes in the SWMR atomic register $PROP[i]$ (line 1). Then, p_i enters a **for** loop (lines 2–6). At step k of the loop, p_i proposes 1 to the underlying binary consensus object $BC[k]$ if p_k has proposed a value; otherwise it proposes 0 (lines 3–4). If $BC[k]$ returns the value 1, p_i returns the value proposed by p_k (lines 5).

Theorem 60 *The construction described in Fig. 14.8 is a wait-free implementation of a multi-valued consensus object from read/write atomic registers and binary consensus objects. Moreover, the values proposed are not required to be bounded.*

Proof The consensus validity property follows directly from the fact that, if $PROP[k]$ is decided by a process, due to the validity of the underlying binary consensus object $BC[k]$, there is a process that has invoked $BC[k]$.bin_propose(1) and before this invocation we had $PROP[k] \neq \bot$.

```
operation $CONS$.mv_propose ($v_i$) is
(1)  $PROP[i] \leftarrow v_i$;
(2)  for $k$ from $1$ to $n$ do
(3)      if ($PROP[k] \neq \bot$) then $bp_i \leftarrow 1$ else $bp_i \leftarrow 0$ end if;
(4)      $res_i \leftarrow BC[k]$.bin_propose($bp_i$);
(5)      if ($res_i = 1$) then return($PROP[k]$) end if;
(6)  end for
end operation.
```

Fig. 14.8 Multi-valued consensus from binary consensus: construction 2



$PROP[\ell] \leftarrow v_\ell$
$PROP[j] \leftarrow v_j$
p_j reads $PROP[j] \neq \perp$
p_j invokes $BC[\ell]$.bin_propose(1)

Fig. 14.9 Linearization order for the proof of the termination property

To prove that *CONS* satisfies the consensus agreement property, let us consider the first binary consensus object ($BC[k]$) that returns the value 1. As processes invoke the binary consensus objects in the same order, it follows from the agreement property of the underlying binary consensus objects that all the processes that invoke $BC[x]$.bin_propose() for $x = 1, ..., (k-1)$ obtain the value 0 from these invocations and the value 1 from $BC[k]$.bin_propose(). As a process exits the loop and decides when it obtains the value 1, it follows that no value different from $PROP[k]$ can be decided.

To prove the termination property (Fig 14.9) of the consensus object *CONS*, let us first observe that, among all the processes that participate (i.e., execute line 1), there is one that is first to write the value it proposes into the array *PROP*. (This follows from the fact that the registers $PROP[1], \ldots, PROP[n]$ are atomic.) Let p_ℓ be that process. Due to the fact that any process p_j writes the value v_j it proposes into $PROP[j]$ before reading any entry of *PROP*, it follows that, when $k = \ell$, it will read $PROP[\ell] \neq \perp$. Hence, all the processes that will invoke $BC[\ell]$.bin_propose(bp) will do so with $bp = 1$. Due to the validity property of $BC[\ell]$, they decide the value 1 from $BC[\ell]$ and consequently exit the **for** loop, which concludes the proof of the termination property of the object *CONS*. □

14.6 Summary

This chapter was devoted to the wait-free implementation of any concurrent object defined by a sequential specification on total operations. It first introduced the notion of a universal construction and the notion of a universal object. It also defined the concept of a consensus object.

The chapter then presented two universal constructions. The first one, which is not bounded, is based on the state machine replication paradigm. The second one, which is bounded, is based on the management of a linked list kept in shared memory. Both constructions rest on consensus objects to ensure that the processes agree on a single order in which their operation invocations appear to have been executed.

The chapter also showed how a multi-valued consensus object can be built from binary consensus objects and atomic read/write registers.

14.7 Bibliographic Notes

- The notions of a universal construction and a universal object are due to M. Herlihy [138]. The wait-free universal construction presented in Sect. 14.4 is due to the same author [138, 139].

  Versions of this time-bounded construction with bounded sequence numbers are described in [138, 172].
- The universal construction based on the state machine replication paradigm presented in Sect. 14.3 is due to R. Guerraoui and M. Raynal [130]. This construction is inspired from a total order broadcast algorithm described in [67]. (The reader interested in the state machine replication paradigm will find more developments on this paradigm in [186, 236, 250].)
- The interested reader will find other constructions in many other papers, e.g., [4, 47, 95, 96].
- A universal construction for large objects is presented in [26].

  Universal constructions for multi-object operations are described in [10, 11, 25]. A multi-object operation is an atomic operation that accesses simultaneously several objects.
- A universal construction in which a process that has invoked an operation on an object can abort its invocation is presented in [77].
- A universal construction suited to transactional memory systems is described in [83].
- The constructions of multi-valued consensus objects from binary consensus objects presented in Sect. 14.5 are the read/write counterparts of constructions designed for message-passing systems described in [217, 236].
- The first universal construction that was presented is based on the state machine replication paradigm and consensus objects. What happens if each consensus instance is replaced by a k-set agreement instance (as defined in Exercise 5, p. 273)? This question is answered in [108]. This answer relies on the observation that, in shared memory systems, solving k-set agreement is the same as solving concurrently k consensus instances where at least one of them is required to terminate [6].

14.8 Exercises and Problems

We will see in Chap. 16 that it is possible to design a consensus object for any number of processes from compare&swap objects or LL/SC registers (this type of register was defined in Sect. 6.3.2). In the following, "directly" means without building intermediate consensus objects.

1. Design a universal construction directly from compare&swap objects (i.e., without building a consensus object from compare&swap objects).
2. Design a universal construction based directly on LL/SC registers.
3. Replace the consensus instances in the first universal construction (Sect. 14.3) and investigate the corresponding properties which are obtained.

   Solution in [108].

Chapter 15
The Case of Unreliable Base Objects

The previous chapter presented universal constructions that allow one to build implementations of any object defined by a sequential specification (on total operations) that are wait-free, i.e., that tolerate any number of process crashes. As we have seen, these constructions rest on two types of objects: atomic read/write registers and consensus objects. These universal constructions assume that these base objects are reliable; namely, they implicitly consider that their behavior always complies with their specification. As an example, given an atomic register R, it is assumed that an invocation of R.read() always returns the last value that was written into R ("last" is with respect to the linearization order). Similarly, given a consensus object *CONS*, an invocation *CONS*.propose() is assumed to always return the single value decided by this consensus object.

This chapter revisits the failure-free object assumption and investigates the case where the base objects are prone to failure. It focuses on the self-implementation of such objects. *Self-implementation* means that the internal representation of the reliable object RO that is built relies on a bounded number m of objects $O_1, \ldots, O_m$ of the very same type. Hence, a reliable atomic register is built from a set of atomic registers of which some (not known in advance) can be faulty, and similarly for a consensus object. Moreover, such a self-implementation has to be *t-tolerant*. This means that the reliability of the object RO that is built has to be guaranteed despite the fact that up to t of the base objects $O_1, \ldots, O_m$ which implement RO can be faulty (Fig. 15.1). Hence, this chapter is devoted to the self-implementation of t-tolerant atomic read/write registers and consensus objects.

From a terminology point of view, wait-freedom is related to process crashes while t-tolerance is related to the failure of base objects.

Keywords Arbitrary failure · Atomic read/write register · Bounded versus unbounded implementation · Consensus object · Crash failure · Graceful degradation · Omission failure · Responsive versus non-responsive failure · Self-implementation · t-Tolerant implementation · Wait-freedom

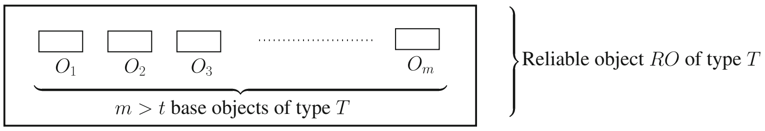

O_1
O_2
O_3
O_m
$m > t$ base objects of type T
Reliable object RO of type T

Fig. 15.1 t-Tolerant self-implementation of an object RO

15.1 Responsive Versus Non-responsive Crash Failures

Intuitively, an object crash failure occurs when the corresponding object stops working. More precisely, two different crash failure models can be distinguished: the *responsive* crash model and the *non-responsive* crash model.

Responsive crashes In the responsive crash failure model, an object fails if it behaves correctly until some time, after which every operation returns the default value $\perp$. This means that the object behaves according to its sequential specification until it crashes (if it ever crashes), and then satisfies the property "once $\perp$, forever $\perp$". The responsive crash model is sometimes called the *fail-stop* model.

Non-responsive crashes In the non-responsive crash model, an object does not return $\perp$ after it has crashed. There is no response after the object has crashed and the invocations of object operations remain pending forever. The non-responsive crash model is sometimes called the *fail-silent* model.

Facing non-responsive failures is more difficult than facing responsive failures. Indeed, in the asynchronous computation model, a process that invokes an operation on an object that has crashed and is not responsive has no means to know whether the object has indeed crashed or is only very slow. As we will see, some objects, which can be implemented in the responsive failure model, can no longer be implemented in the non-responsive failure model.

A lower bound It is easy to see that $t + 1$ is a lower bound on the number of base objects $O_1, \ldots O_m$ required to mask up to t faulty base objects. If an operation on the constructed object RO accesses only t base objects, and all of them fail, there is no way for the constructed object to mask the base object failures.

15.2 SWSR Registers Prone to Crash Failures

This section presents self-implementations of a wait-free t-tolerant SWSR register from $m > t$ SWSR registers prone to responsive and non-responsive crash failures. For responsive crash failures $m = t + 1$, while $m = 2t + 1$ for non-responsive crash failures. As far as MRMW registers are concerned, appropriate constructions have been presented in Chaps. 11 and 13.

15.2.1 Reliable Register When Crash Failures Are Responsive: An Unbounded Construction

As it is based on sequence numbers, this first construction is unbounded. It uses $t+1$ underlying SWSR atomic registers denoted $REG[1..(t+1)]$. Each register $REG[i]$ is made up of two fields denoted $REG[i].sn$ (sequence number part) and $REG[i].val$ (value part). Each $REG[i]$ is initialized to the pair $(v_0, 0)$, where v_0 is the initial value of the constructed register.

The algorithm implementing the operation RO.write(v) The algorithms implementing the read and write operations of the t-tolerant SWSR register RO are described in Fig. 15.2. An invocation of RO.write(v) writes the pair, made up of the new value plus its sequence number, in all the base registers. The local variable sn of the writer is used to generate sequence numbers (its initial value being 0).

The algorithm implementing the operation RO.read() The reader keeps in a local variable denoted $last$, initialized to $(v_0, 0)$, a copy of the pair (v, sn) with the highest sequence number it has ever seen. This variable allows prevention of new/old inversions when base registers or the writer crash.

An invocation of RO.read() consists in reading the base registers (in any order). Let us observe that, as at most t registers may crash, at least one register returns always a non-$\perp$ value. If the reader reads a more recent value from a base register whose read returns a non-$\perp$ value, it updates $last$ accordingly. Finally, it returns the value $last.val$, i.e., the value associated with the highest sequence number it has ever seen ($last.sn$).

Remark It is important to notice that the read and write operations on RO access the base registers in any order. This means that no operation invocation on a base register depends on a previous operation invocation on another base register. Said in another way, these invocations could be issued in parallel, thereby favoring efficiency.

```
operation $RO$.write($v$) is % invoked by the writer %
    $sn \leftarrow sn + 1$;
    for $j \in \{1, \ldots, t+1\}$ do $REG[j] \leftarrow \langle v, sn \rangle$ end for;
    return()
end operation.

operation $RO$.read() is % invoked by the reader %
    for $j \in \{1, \ldots, t+1\}$ do
        $aux \leftarrow REG[j]$;
        if $(aux \neq \perp) \wedge (aux.sn > last.sn)$ then $last \leftarrow aux$ end if
    end for;
    return($last.val$)
end operation.
```

Fig. 15.2 t-Tolerant SWSR atomic register: unbounded self-implementation (responsive crash)

Let us observe that the version of the construction with such parallel invocations is optimal as far as time complexity is concerned.

Theorem 61 *The construction described in Fig. 15.2 is a wait-free t-tolerant self-implementation of an SWSR atomic register from $t + 1$ SWSR atomic registers that may suffer responsive crash failures.*

Proof As already noticed, the construction is trivially wait-free. Moreover, as by assumption there is at least one base register that does not crash, each invocation of RO.read() returns a non-$\perp$ value and consequently the register RO is reliable. So, it remains to show that it behaves as an atomic register. This is done by (a) first defining a total order on the invocations of RO.write() and RO.read(), and (b) then showing that the resulting sequence satisfies the sequential specification of a register. This second step uses the fact that there exists a total order on the accesses to the base registers (as those registers are atomic).

Let us associate with each invocation of RO.write() the sequence number of the value it writes. Similarly, let us associate with each invocation of RO.read() the sequence number of the value it reads. Let $\widehat{S}$ be the total order on the invocations of RO.write() and RO.read() defined as follows. The invocations of RO.write() are ordered according to their sequence numbers. Each invocation of RO.read() whose sequence number is sn is ordered just after the invocation of RO.write() that has the same sequence number. If two or more invocations of RO.read() have the same sequence number, they are ordered in $\widehat{S}$ according to the order in which they have been issued by the reader. We have the following:

- It follows from its definition that $\widehat{S}$ includes all the operation invocations issued by the reader and the writer (except possibly their last operation if they crash).
- Due to the way the local variable sn is used by the writer, the invocations of RO.write() appear in $\widehat{S}$ in the order they have been issued by the writer.
- Similarly, the invocations of RO.read() appear in $\widehat{S}$ according to the order in which they have been issued by the reader. This is due to the local variable $last$ used by the reader (the reader returns the value with the highest sequence number it has ever obtained from a base register).
- As the base registers are atomic, the base operations on these registers are totally ordered. Consequently, when we consider this total order, a base read operation that obtains the sequence number sn from an atomic register $REG[j]$ appears after the base write operation that wrote sn into that register.

  As $\widehat{S}$ is such that an invocation of RO.read() that obtains a value whose sequence number is sn appears after the snth and before the $(sn + 1)$th invocation of RO.write(), it follows that $\widehat{S}$ is consistent with the occurrence order defined by the read and write invocations on the base objects.

It follows from the previous observations that $\widehat{S}$ is a correct linearization of the invocations of RO.read() and RO.write(). Consequently, the constructed register RO is atomic. □

15.2.2 Reliable Register When Crash Failures Are Responsive: A Bounded Construction

Eliminating sequence numbers When we consider the previous construction, an interesting question is the following: Is it possible to design a t-tolerant SWSR atomic register from $t+1$ bounded base registers; i.e., are the sequence numbers necessary? The following construction that is based on bounded base registers shows that they are not. Moreover, that construction is optimal in the sense that each base register has only to contain the value that is written. No additional control information is required.

The read and write algorithms The corresponding construction is described in Fig. 15.3. The writer simply writes the new value in each base register, in increasing order, starting from $REG[1]$ until $REG[t+1]$. The reader scans sequentially the registers in the opposite order, starting from $REG[t+1]$. It stops just after the first read of a base register that returns a non-$\perp$ value. As at least one base register does not crash (model assumption), the reader always obtains a non-$\perp$ value. (Let us recall that, as we want to build a t-tolerant read/write register, the construction is not required to provide guarantees when more than t base objects crash.)

It is important to remark that, differently from the construction described in Fig. 15.2, both the operations RO.write() and RO.read() now have to follow a predefined order when they access the base registers $REG[1..(t+1)]$. Moreover, the order for reading and the order for writing are opposite. These orders are depicted in Fig. 15.4 with a space–time diagram in which the "time lines" on the base registers is represented. A black circle indicates an invocation of a read or write on a base register $REG[k]$. An invocation of RO.read() stops reading base registers as soon as it reads a non-$\perp$ value.

Why read and write operations have to scan the base registers in reverse order To understand why the operations RO.write(v) and RO.read() have to access the base registers in opposite order, let us consider the following scenario where both operations access the base registers in the same order, e.g., from $REG[1]$ to

```
operation $RO$.write($v$) is % invoked by the writer %
    for $j$ from $1$ to $t+1$ do $REG[j] \leftarrow v$ end for;
    return()
end operation.

operation $RO$.read()is % invoked by the reader %
    for $j$ from $t+1$ to $1$ do
        $res \leftarrow REG[j]$;
        if $(res \neq \perp)$ then return($res$) end if
    end for
end operation.
```

Fig. 15.3 t-Tolerant SWSR atomic register: bounded self-implementation (responsive crash)

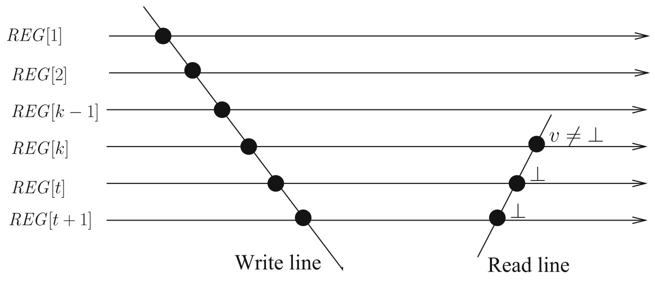


Fig. 15.4 Order in which the operations access the base registers

REG$[t+1]$. The write updates *REG*$[1]$ to x and crashes just after. Then, an invocation of RO.read() obtains the value x. Sometime later, *REG*$[1]$ crashes. After that crash occurs, the reader reads *REG*$[1]$, obtains $\perp$, then reads *REG*$[2]$ and obtains y, a value that was written before x. Consequently, the reader suffer, a new/old inversion and RO is not atomic. Forcing the reader to access the base registers in the reverse order (with respect to the writer) ensures that, if the reader returns v from *REG*$[j]$, all the base registers *REG*$[k]$ such that $j < k \leq t+1$ have crashed. More generally, as we have seen previously, if the reader and the writer do not access the base registers in opposite order, additional control information has to be used (to create some order) such as sequence numbers.

Remark Scanning base registers in one direction when writing and in the other direction when reading is a technique that has already been used in Sect. 11.3 devoted to the construction of b-valued regular (atomic) registers from regular (atomic) bits.

Bounded versus unbounded construction It is interesting to emphasize the tradeoff between this bounded construction and the previous unbounded one. When the base registers are accessed in parallel, the unbounded construction described in Fig. 15.2 is time-optimal but requires additional control information, namely sequence numbers. Differently, the bounded construction described in Fig. 15.3 is space-optimal (no additional control information is required) but requires sequential invocations on the base registers.

Theorem 62 *The construction described in Fig. 15.3 is a wait-free t-tolerant self-implementation of an SWSR atomic register from $t+1$ SWSR atomic registers that may suffer responsive crash failures. Moreover, it is space-optimal.*

Proof The wait-freedom property follows directly from the fact that the loops are bounded. Due to the assumption that at most t base registers may crash, the value returned by an invocation of RO.read() is a value that was previously written. Consequently, the constructed object is a reliable register.

The proof that the constructed register RO is atomic is done incrementally. It is first shown that the register is safe, then regular, and finally atomic. The proof for going from regularity to atomicity consists in showing that there is no new/old inversion, from which atomicity follows from Theorem 43 in Chap. 11 (this theorem states that any execution of a regular register in which there is no new/old inversion is linearizable and hence atomic).

- Safeness property. Let us consider an invocation of RO.read() when there is no concurrent invocation of RO.write(). Safeness requires that, in this scenario, the read returns the last value that was written into RO.

  As (by assumption) there is no concurrent write on RO, we conclude that the writer has not crashed during the last invocation of RO.write() issued before the invocation of RO.read() (otherwise, this write invocation would not be terminated and consequently would be concurrent with the read invocation).

  The last write updated all the non-crashed registers to the same value v. It follows that, whatever the base register from which the read operation obtains a non-$\perp$ value, it obtains and returns the value v, which is the last value written into RO.

- Regularity property. If an invocation of RO.read() is concurrent with one or several invocations of RO.write(), we have to show that the read invocation has to obtain the value of the constructed register before these write invocations or the value written by one of them.

  Let us first observe that an invocation r of RO.read() cannot obtain from a base register a value that has not yet been written into it. We conclude from this observation that a read invocation cannot return a value that has not yet been written by a write invocation.

  Let v be the value of the register before the concurrent invocations of RO.write(). This means that all the non-crashed base registers are equal to v before the first of these concurrent write invocations. If r obtains the value v, regularity is ensured. So, let us assume that r obtains another value v' from some register $REG[x]$. This means that $REG[x]$ has not crashed and was updated to v' after having been updated to v. This can only be done by a concurrent write invocation that writes v' and was issued by the writer after the write of v. The constructed register is consequently regular.

- Atomicity property. We prove that there is no new/old inversion. Let us assume that two invocations of RO.read(), say r_1 and r_2, are such that r_1 is invoked before r_2, r_1 returns $v2$ that was written by w_2, r_2 returns $v1$ that was written by w_1, and w_1 was issued before w_2 (Fig. 15.5).

  The invocation r_1 returns $v2$ from some base register $REG[x]$. It follows from the read algorithm that all the base registers $REG[y]$ such that $x < y \leq t + 1$ have

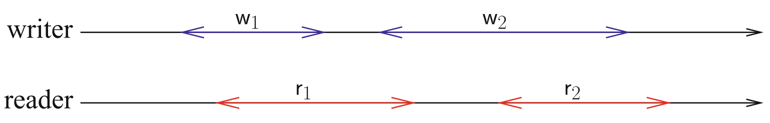


Fig. 15.5 Proof of the "no new/old inversion" property

```
operation $RO$.read() is
    for $j$ from $shortcut$ to $1$ do
        $aux \leftarrow REG[j]$;
        if $(aux \neq \perp)$ then $shortcut \leftarrow j$; return$(aux)$ end if
    end for
end operation.
```

Fig. 15.6 A simple improvement of the bounded construction

crashed. It also follows from the write algorithm that the non-crashed registers from $REG[1]$ to $REG[x-1]$ contain $v2$ or a more recent value when r_1 returns $v2$.

As the base registers from $REG[t+1]$ until $REG[x+1]$ have crashed when r_2 is invoked, that read invocation obtains $\perp$ from all these registers. When it reads the atomic register $REG[x]$, it obtains either $v2$, or a more recent value, or $\perp$.

- If r_2 obtains $v2$ or a more recent value, there is no new/old inversion.
- If r_2 obtains $\perp$, it continues reading from $REG[x-1]$ until it finds a base register $REG[y]$ $(y < x)$ from which it obtains a non-$\perp$ value. Also, as the write algorithm writes the base registers in increasing order starting from $REG[1]$, it follows that no register from $REG[1]$ until $REG[x-1]$ (which is not crashed when it is read by r_2) can contain a value older than $v2$; namely, it can only contain $v2$ or a more recent value. It follows that there is no possibility of new/old inversion also in this case. $\square$

A simple improvement An easy way to improve the time efficiency of the previous operation RO.read() consists in providing the reader process with a local variable (denoted *shortcut* and initialized to $t+1$) that keeps an array index such that, to the reader's knowledge, each $REG[k]$ has crashed for $shortcut < k \leq t+1$. The resulting read algorithm is described in Fig. 15.6. It is easy to see that, if after some time no more base registers crash, *shortcut* always points to the first (in descending order) non-crashed base register. This means that there is a time after which the duration of a read operation is constant in the sense that it depends neither on t nor on the number of base registers that have actually crashed.

15.2.3 *Reliable Register When Crash Failures Are Not Responsive: An Unbounded Construction*

When crash failures are not responsive, the construction of an SWSR atomic register RO is still possible but requires a higher cost in terms of base registers; namely, $m \geq 2t+1$ base registers are then required.

```
operation $RO$.write($v$) is
    $sn \leftarrow sn + 1$;
    concurrently for each base register $j \in \{1, \ldots, m\}$
                    do issue write $(v, sn)$ into $REG[j]$ end for;
    wait (a majority of the previous base write invocations have terminated);
    return()
end operation.

operation $RO$.read() is
    concurrently for each base register $j \in \{1, \ldots, m\}$
                    do issue read () on $REG[j]$ end for;
    wait (a majority of the previous base read invocations have terminated);
    let $pairs$= the set of pairs $\langle val, sn \rangle$ received from the previous read invocations;
    $last \leftarrow$ the pair in the set $pairs \cup \{last\}$ with the highest sequence number;
    return($last.val$)
end operation.
```

Fig. 15.7 t-Tolerant SWSR atomic register: unbounded self-implementation (non-responsive crash)

Base principles The base principles of such a construction are relatively simple. They are the following:

- The use of sequence numbers (as in the construction for responsive failures, Fig. 15.2). These sequence numbers allow the most recent value written in RO to be known.
- The use of the majority notion. As the model assumes that, among the m base registers, at most t can be faulty, taking $m > 2t$ ensures that there is a majority of base registers that do not crash. Said differently, any set of $t + 1$ base registers contains at least one register which is not faulty.
- The parallel activation of read operations on base registers. This allows one to cope with the fact that some read of base registers are not responsive. Combined with the majority of correct base registers, this ensures that invocations of RO.read() do not remain blocked forever.

An unbounded construction The construction described in Fig. 15.7 is a straightforward extension of the algorithm described in Fig. 15.3. It additionally takes into account the fact that some base read/write invocations might never answer.

So, considering $m = 2t + 1$, the construction issues read (or write) invocations in parallel on the base registers $REG[1], \ldots, REG[2t+1]$, in order to prevent permanent blocking, which could occur if the base operations were issued sequentially. As in the algorithm described in Fig. 15.3, the reader maintains a local variable $last$ that keeps the (val, sn) pair with the highest sequence number it has ever read from a base register.

This construction shows that, when one is interested in building a reliable SWSR atomic register, the price to pay to go from responsive failures to non-responsive failures is to increase the number of base registers from $t + 1$ to $2t + 1$.

from $t + 1$ base registers to $2t + 1$ base registers.

Theorem 63 *The construction described in Fig. 15.7 is a wait-free t-tolerant self-implementation of an SWSR atomic register from $m = 2t + 1$ base SWSR atomic registers that can suffer non-responsive crash failures.*

Proof (Sketch) The proof is a simple adaptation of the proof of Theorem 62 to the context of non-responsive crash failures. It is left to the reader as an exercise. (The fact that at least one non-faulty base register is written (read) used in Theorem 62 is replaced here by the assumption that a majority of base registers do not crash.) □

15.3 Consensus When Crash Failures Are Responsive: A Bounded Construction

This section presents a wait-free t-tolerant self-implementation of a consensus object R_CONS built from $m = t + 1$ base consensus objects prone to responsive crash failures. As for registers, it is easy to see that $t + 1$ is a tight lower bound on the number of crash-prone base consensus objects.

15.3.1 The "Parallel Invocation" Approach Does Not Work

Before presenting a construction that builds a t-tolerant consensus object, let us give an intuitive explanation of the fact that the "replicated state machine with parallel invocations" approach does not work. This approach considers copies of the object (here base consensus objects) on which the same operation is applied in parallel.

So, assuming $m = 2t + 1$ base consensus objects $CONS[1..m]$, let us consider that the algorithm implementing $R\_CONS.\mathsf{propose}(v)$ is implemented as follows: the invoking process (1) invokes in parallel $CONS[k].\mathsf{propose}(v)$ for $k \in \{1, \ldots, m\}$ and then (2) takes the value decided by a majority of the base consensus objects.

As there is a majority of base objects that are reliable, this algorithm does not block, and the invoking process receives decided values at least from a majority of base consensus objects. But, according to the values proposed by the other processes, it is possible that none of the values it receives is a majority value. It is even possible that it receives a different value from each of the $2t + 1$ base consensus objects if there are $n \geq m = 2t + 1$ processes and all have proposed different values to the consensus object R_CONS.

While the "parallel invocation" approach works for objects such as atomic read/write registers (see above), it does not work for consensus objects. This comes from the fact that registers are *data* objects, while consensus are *synchronization* objects, and synchronization is inherently non-deterministic.

```
operation $R\_CONS$.propose($v$) is
(1) $est_i \leftarrow v$;
(2) for $k$ from $1$ to $t+1$ do
(3)         $aux \leftarrow CONS[k]$.propose($est_i$);
(4)         if $(aux \neq \perp)$ then $est_i \leftarrow aux$ end if
(5) end for;
(6) return($est_i$)
end operation.
```

Fig. 15.8 Wait-free t-tolerant self-implementation of a consensus object (responsive crash/omission)

15.3.2 A t-Tolerant Wait-Free Construction

The $t + 1$ base consensus objects are denoted $CONS[1..(t + 1)]$. The construction is described in Fig. 15.8. The variable est_i is local to the invoking process and contains its current estimate of the decision value. When a process p_i invokes R_CONS.propose(v), it first sets est_i to the value v it proposes. Then, p_i sequentially visits the base consensus objects in a predetermined order (e.g., starting from $CONS[1]$ until $CONS[t + 1]$; the important point is that the processes use the same visit order). At step k, p_i invokes $CONS[k]$.propose(est). Then, if the value it obtains is different from $\perp$, p_i adopts it as its new estimate value est_i. Finally, p_i decides the value of est after it has visited all the base consensus objects. Let us observe that, as at least one consensus object $CONS[\ell]$ does not crash, all the processes that invoke $CONS[\ell]$.propose() obtain the same non-$\perp$ value from that object.

Theorem 64 *The construction described in Fig. 15.8 is a wait-free t-tolerant self-implementation of a consensus object from $t + 1$ base consensus objects that can suffer responsive crash failures.*

Proof The proof has to show that, if at most t base consensus objects crash, the object that is built satisfies the validity, agreement and wait-free termination properties of consensus.

As any base consensus object $CONS[k]$ is responsive, it follows that any invocation of $CONS[k]$.propose() terminates (line 3). It follows that, when executed by a correct process, the **for** loop always terminates. The wait-free termination follows directly from these observations.

When a process p_i invokes R_CONS.propose(v), it first initializes its local variable est_i to the value v it proposes. Then, if est_i is modified, it is modified at line 4 and takes the value proposed by a process to the corresponding base consensus object. By backward induction, that value was proposed by a process, and the consensus validity property follows.

Let $CONS[x]$ be the first (in the increasing order on x) non-faulty base consensus object (by assumption, such a base object exists). Let v be the value decided by that consensus object. It follows from the agreement property of $CONS[x]$ that all the processes that invoke $CONS[x]$.propose(est) decide v. It then follows that, from then on, only v can be proposed to the base consensus objects $CONS[x+1]$ until $CONS[t+1]$. Consequently, v is the value decided by the processes that execute line 6, and the agreement property of the constructed consensus object *R_CONS* follows. (As we can see, the fact that all the processes "visit" the base consensus objects in the same order is central to the proof of the agreement property.) □

15.3.3 Consensus When Crash Failures Are Not Responsive: An Impossibility

This section presents an impossibility result. Differently from atomic registers, no t-tolerant consensus object can be built from crash-prone non-responsive consensus objects.

Theorem 65 *There is no wait-free t-tolerant self-implementation of a consensus object from crash-prone non-responsive consensus objects.*

Proof The proof is left to the reader. It constitutes Exercise 2. □

15.4 Omission and Arbitrary Failures

The ideas and algorithms developed in this section are mainly due to P. Jayanti, T.D. Chandra, and S. Toueg (1998).

15.4.1 Object Failure Modes

We have seen in the previous sections the notions of responsive and non-responsive object failures. With responsive failures, an invocation of an object operation always returns a response but that response can be $\perp$. With non-responsive failures, an invocation of an object operation may never return. Except for a few objects (including atomic registers) it is usually impossible to design a t-tolerant wait-free implementation of objects when failures are non-responsive (e.g., consensus objects).

Failure modes In addition to the responsiveness dimension of object failures, three modes of failure can be defined. These modes define a second dimension with respect to object failures:

- Crash failure mode. An object experiences a crash if there is a time τ such that the object behaves correctly up to time τ, after which all its operation invocations return the default value $\perp$. As we have seen, this can be summarized as "once $\perp$, forever$\perp$". This failure mode was investigated in Sects. 15.1–15.3 for atomic read/write registers and consensus objects.

- Omission failure mode. An object experiences an omission failure with respect to a process p if, after some time, it behaves as a crash object with respect to p. When failures are responsive, this means that, after some finite time, it always returns $\perp$ to any operation invocation issued by p.
  Let us observe that an object can commit omission failures with respect to some processes p_x, p_y, etc., but behave correctly with respect to other processes p_z, etc.
- Arbitrary failure mode. An object experiences arbitrary failures if, after some time, it behaves arbitrarily; i.e., its behavior does not follow its specification. When failures are responsive, this means that any operation returns a response but this response is arbitrary.

t-Tolerance with respect to a failure mode An implementation of an atomic concurrent object O is t-tolerant with respect to a failure mode $\mathcal{F}$ (crash, omission, or arbitrary) if, despite the occurrence of up to t base objects that fail according to the mode $\mathcal{F}$,

- The object O remains correct (i.e., its executions remain linearizable), and
- Its operations are wait-free.

Section 15.2 presented a t-tolerant self-implementation of an atomic read/write register for the crash failure mode (with responsive and non-responsive failures), while Sect. 15.3 presented a t-tolerant self-implementation of a consensus object for the crash failure mode (with responsive failures).

Hierarchy (severity) of failure modes Let us observe that an object implementation that is t-tolerant with respect to the omission (arbitrary) failure mode is also t-tolerant with respect to the crash (omission) failure mode. This observation defines two hierarchies of failure modes (one for responsive failures and the other one for non-responsive failures), namely

$$\text{crash} \subset \text{omission} \subset \text{arbitrary}.$$

Considering an atomic object O for which we have an implementation $I(O)$ that is t-tolerant with respect to the omission (arbitrary) failure mode, it may be possible to design an implementation $I'(O)$ that (a) is t-tolerant with respect to the crash (omission) failure mode only and (b) is more (time or space) efficient than $I(O)$. More generally, the cost of a t-tolerant object implementation usually depends on the failure mode that is considered.

On the meaning of t-tolerance with respect to a failure mode Let us notice that an object implementation I that is t-tolerant with respect to a failure mode $\mathcal{F}$ is required to work only if (a) at most t base objects fail and (b) they fail according to the mode $\mathcal{F}$.

This means that I is not required to work correctly if more than t base objects fail according to the mode $\mathcal{F}$ or if the failure of one or more objects does not belong to the failure mode $\mathcal{F}$. This is because, when this occurs, the implementation does not execute in the computation model for which it was designed.

15.4.2 Simple Examples

This section presents two simple t-tolerant object self-implementations. The first considers the omission failure mode, while the second considers the arbitrary failure mode.

A t-tolerant self-implementation of consensus for responsive omission failures Let us consider the t-tolerant self-implementation of a consensus object for the responsive crash failure mode described in Fig. 15.8. It is easy to see that this algorithm still works when the failure mode is omission (instead of crash). This is due to the fact that, due to definition of t, at least one of the $(t + 1)$ sequential iterations necessarily uses a correct base consensus object.

A t-tolerant self-implementation of an SWSR safe register for responsive arbitrary failures Responsive arbitrary failure of a base read/write register *BR* means that a read of *BR* can return an arbitrary value (even when there is no concurrent write) and a write of a value v into *BR* can actually deposit any value $v' \neq v$.

The t-tolerant SWSR safe register *SR* is built from $m = 2t + 1$ base safe registers *REG*$[1..m]$. Each base register is initialized to the initial value of *SR*. The self-implementation of RO for responsive arbitrary failures is described in Fig. 15.9. The algorithm implementing *SR*.write(v) consists in writing the value v in each base safe register. When the reader invokes *SR*.read() it first reads all the base safe registers (line 3) and then returns the value that is the most present in *REG*$[1..m]$ (lines 4–5; if several values are equally most present, any of them is chosen).

Theorem 66 *The construction of Fig. 15.9 is a bounded wait-free t-resilient implementation of an SWSR safe register.*

Proof The fact that the construction is bounded follows directly from an examination of the algorithm. Moreover, as the read and write operations on the base registers are responsive, the construction is wait-free.

Let us assume that at most t base registers fail. This means that at most t base registers can contain arbitrary values. This means that, if *SR*.read() is invoked while

```
operation $SR$.write($v$) is
 (1)  for $k \in \{1,\ldots,2t+1\}$ do $REG[k] \leftarrow v$ end for;
 (2)  return()
end operation.

operation $SR$.read() is
 (3)  for $k \in \{1,\ldots,2t+1\}$ do $reg[k] \leftarrow REG[k]$ end for;
 (4)  $res \leftarrow$ most present value in $reg[1..2t+1]$;
 (5)  return($res$)
end operation.
```

Fig. 15.9 Wait-free t-tolerant (and gracefully degrading) self-implementation of an SWSR safe register (responsive arbitrary failures)

the writer does not execute SR.write(), it obtains at least $t+1$ copies of the same value v (which is the value written by the last invocation of SR.write()). As $t+1$ defines a majority, the value v is returned. Finally, the fact that any value can be returned when SR.read() is concurrent with an invocation of SR.write() concludes the proof of the theorem. $\square$

Let us remember that, due to the suite of constructions presented in Part V of this book, it is possible to build an MWMR atomic register from reliable safe bits. Hence, stacking these constructions on top of the previous one, it is possible to build an omission-tolerant wait-free MWMR atomic register from safe bits prone to responsive arbitrary failures.

15.4.3 Graceful Degradation

As far as fault-tolerance is concerned, an important issue is the following one: How does a wait-free t-tolerant implementation of an object behave when more than t base objects fail (i.e., in the executions that are outside the system model)? Does the object that is built fail in the same failure mode as its base objects or does it fail in a more severe failure mode? Let us observe that, in the second case, we have failure amplification: the object that is built fails more severely than the base objects from which it is built! The notion of graceful degradation was introduced to better understand and answer this question.

Definition An implementation of an object O is *gracefully degrading* if O never fails more severely than the base objects from which it built, whatever the number of these base objects that fail.

As an example, let us consider that the base objects from which O is implemented can fail by crashing. We have the following:

- If an implementation I of O is wait-free and correct despite the crash of any number of processes and the crash of up to t base objects, then I is t-tolerant for the crash failure mode.
- If an implementation I of O is wait-free and fails only by crash, despite the crash of any number of processes and the crash of any number of (i.e., more than t) base objects, I is gracefully degrading.

Hence, a t-tolerant (with respect to a failure mode $\mathcal{F}$) gracefully degrading wait-free implementation is a t-tolerant implementation that never fails more severely than the failure mode $\mathcal{F}$, when more than t base objects fail according to $\mathcal{F}$.

As an example let us consider the implementation of a consensus object described in Fig. 15.8. Theorem 64 has shown that this implementation is wait-free and t-tolerant with respect to the responsive crash failure mode. It is easy to see that it is not gracefully degrading. To that end let us consider an execution in which the

$(t+1)$ base consensus objects fail by crash from the very beginning. It follows that, for any k, all the invocations $CONS[k]$.propose() return $\perp$. Hence, the local variable est_i of each process p_i remains forever equal to the value proposed by p_i, and consequently each process decides its initial value. It follows that, when more than t base consensus objects fail, the consensus object that is built can fail according to the arbitrary failure mode.

A gracefully degrading t-tolerant self-implementation of a safe SWSR read/write register (with respect to arbitrary failures) The construction described in Fig. 15.9 is a gracefully degrading t-tolerant self-implementation for the arbitrary failure mode. This means that, when more than t base safe registers experience arbitrary failures, the constructed object fails according to the arbitrary failure mode. This follows directly from the fact that this construction is t-tolerant (Theorem 66) and the arbitrary failures define the most severe failure mode.

Consensus with responsive omission failures: specification As we have seen, responsive omission failure means that an invocation of an object operation may return the default value $\perp$ instead of returning a correct value.

The definition of a consensus object (which is a one-shot object) has to be re-visited to take into account omission failures. The corresponding weakened definition is as follows:

- Validity. A decided value v is a proposed value or the default value $\perp$. Moreover, it is not $\perp$ if the consensus object does not suffer an omission failure.
- Integrity. A process decides at most once.
- Agreement. If a process decides $v \neq \perp$ and another process decides $w \neq \perp$, then $v = w$.
- Termination. An invocation of propose() by a correct process terminates.

As we can see, taking into account omission failures requires one to modify only the validity and agreement properties. It is easy to see that this definition boils down to the usual definition if we eliminate the possibility of deciding $\perp$.

A gracefully degrading t-tolerant self-implementation of consensus (with respect to omission failures) A wait-free t-tolerant (with respect to responsive omission failures) gracefully degrading self-implementation of a consensus object is described in Fig. 15.10. This construction is due to P. Jayanti, T.D. Chandra and S. Toueg (1999). It uses $m = 2t+1$ base consensus objects which are kept in the array $CONS[1..m]$.

In this construction, each process p_i manages a local variable est_i which contains its estimate of the decision value and an array $dec_i[1..(2t+1)]$ that it will use to take its final decision. Similarly to the construction described in Fig. 15.8 (which is t-tolerant with respect to responsive omission failures but not gracefully degrading), a process p_i visits sequentially all the base consensus objects. The main differences between the construction of Fig. 15.8 (which is not gracefully degrading) and the one of Fig. 15.10 (which is designed to be gracefully degrading) are:

```
operation $R\_CONS$.propose($v$) is
(1)  $est_i \leftarrow v$;
(2)  for $k$ from $1$ to $2t+1$ do
(3)      $dec_i[k] \leftarrow CONS[k]$.propose($est_i$);
(4)      if $(dec_i[k] \neq \bot) \wedge (dec_i[k] \neq est_i)$
(5)        then $est_i \leftarrow dec_i[k]$;
(6)             $dec_i[1..(k-1)] \leftarrow [\bot, \ldots, \bot]$
(7)      end if
(8)  end for;
(9)  if $(\#_\bot(dec_i) > t)$ then $est_i \leftarrow \bot$ end if;
(10) return($est_i$)
end operation.
```

Fig. 15.10 Gracefully degrading self-implementation of a consensus object (responsive omission)

- The number of base consensus objects ($2t + 1$ instead of $t + 1$), and
- A more sophisticated local processing after an invocation of $CONS[k]$.propose (est$_i$) (invoked at line 3).

More specifically, when p_i executes its kth iteration, the value v returned by $CONS[k]$.propose(est_i) is saved in the local variable $dec_i[k]$ (line 3). Then (line 4), if $v \neq \bot$, we can conclude that $CONS[k]$ has not committed an omission failure with respect to p_i. Moreover, if $v \neq est_i$, we can deduce that each of the base objects $CONS[1..(k-1)]$ has committed a failure (if one of them, say $CONS[x]$, was correct, the processes would have obtained the value decided by that object during iteration $x \leq k - 1$).

Hence, if v is different from both $\bot$ and est_i, p_i considers v as its new estimate (line 5) and sets accordingly the $(k - 1)$ first entries of its local array dec_i to $\bot$ (line 6). It follows that, at the end of the kth iteration, the only values that $dec_i[1..k]$ can contain are $\bot$ and the current value v of est_i.

Finally, when it has exited the **for** loop (lines 2–8), p_i computes its decision from the current value of the array $dec_i[1..(2t+1)]$. If more than t entries of dec_i contain $\bot$, p_i decides $\bot$ and, consequently, the constructed object R_CONS fails by omission with respect to p_i. Otherwise at least $t + 1$ entries of dec_i contain the same value $v \neq \bot$ and we have $est_i = v$. In that case, p_i decides v.

Theorem 67 *The construction described in Fig. 15.10 is a gracefully degrading wait-free t-tolerant implementation of a consensus object from* $(2t + 1)$ *base consensus objects prone to responsive omission failures.*

Proof Wait-freedom. This property follows directly from the fact that the **for** loop is bounded and, as failures are responsive, each invocation $CONS[k]$.propose(est_i) terminates.

To prove the consensus validity property and the graceful degradation property, we have to show that the value that is decided by a process (i) is either a proposed value or $\perp$ and (ii) is not $\perp$ when no more than t base consensus objects fail.

- Part (i). Let us first observe that each local variable est_i is initialized to a proposed value (line 1). Then, it follows from (a) the validity of the omission-prone base consensus objects (a decided value is a proposed value or $\perp$) and (b) a simple induction on k that the values written in $dec_i[1..(2t+1)]$ are only $\perp$ or the value of some est_j. As the value decided by p_i is a value stored in $dec_i[1..(2t+1)]$ (lines 3–5) part (i) of the consensus validity property follows.
- Part (ii). By assumption, there are c correct base consensus objects where $t+1 \leq c \leq 2t+1$. Let $CONS[k_1]$, $CONS[k_2], \ldots, CONS[k_c]$ be this sequence of correct base objects. As there are at most t faulty base consensus objects, we have $k_1 \leq t+1 \leq c \leq 2t+1$.

  As $CONS[k_1]$ is correct and no process proposes $\perp$, $CONS[k_1]$ returns the same value v to all the processes that invoke $CONS[k_1]$.propose(). Hence, each non-crashed process p_i is such that $est_i = v$ at the end of its k_1th loop iteration. From then on, as (a) a base object can fail only by omission (it then returns $\perp$ instead of returning a proposed value) and (b) due to their agreement property each of the $c-1$ remaining correct consensus objects return v, it follows that, for any x, $k_1 < x \leq 2t+1$, the predicate of line 4 $(dec_i[x] \neq \perp, est_i)$ is never satisfied. Consequently, lines 5–6 are never executed after the k_1th loop iteration.

  Moreover, as there are $c \geq t+1$ consensus objects that do not fail, it follows that, at the end of the last iteration, at least c entries of the array dec_i contain the value v, the other entries containing $\perp$. It follows that, when a process exits the loop, the predicate at line 9 cannot be satisfied. Consequently, p_i decides the value $v \neq \perp$ kept in est_i, which concludes the proof of part (ii).

To prove the consensus agreement property we have to show that, if a process p_i decides $v \neq \perp$ and a process p_j decides $w \neq \perp$, then $v = w$.

The proof is by contradiction. Let us assume that p_i decides $v \neq \perp$ while p_j decides $w \neq \perp$ and $w \neq v$. As p_i decides v, it follows from line 9 that at least $(t+1)$ entries of $dec_i[1..(2t+1)]$ contain v. Similarly, at least $(t+1)$ entries of $dec_j[1..(2t+1)]$ contain w. As there are only $(2t+1)$ base consensus objects, it follows from the previous observation that there is a base consensus object $CONS[k]$ such that the invocations of $CONS[k]$.propose() by p_i and p_j returned $v \neq \perp$ to p_i and $w \neq \perp$ to p_j. But this is impossible because, as $CONS[k]$ can fail only by omission, it cannot return different different non-$\perp$ values to distinct processes. It follows that $v = w$. $\square$

15.4.4 Fault-Tolerance Versus Graceful Degradation

A general result An important result proved by P. Jayanti, T.D. Chandra, and S. Toueg (1998) is the fact that, while there is a t-tolerant wait-free implementation of consensus objects for the responsive crash failure mode (Fig. 15.8), no such implementation I can be gracefully degrading. This means that, whatever I, if more than t base consensus objects crash, the constructed consensus object can exhibit omission or arbitrary failures.

It follows that, while t-tolerant gracefully degrading implementations exist for the responsive omission (or arbitrary) failure mode for many objects, there are objects (such as consensus objects) that have t-tolerant wait-free implementations for any failure mode but do not have a t-tolerant gracefully degrading implementation for responsive crash failures. When more than t base objects fail by crash, their failures give rise to a more severe failure at the level of the constructed object. This phenomenon is called *failure amplification*.

If an implementation of an object O is gracefully degrading for the responsive omission failure mode, base objects can experience "severe" failures (namely, omission failures) but the object O can also experience "severe" failures (omissions). The fact that the considered failure mode for the base objects is "severe" makes it possible to design a gracefully degrading implementation because O also is allowed to experience the same type of severe failures. If an implementation has to be gracefully degrading for responsive crash failures, base objects can experience benign failures (crash) but, when more than t base objects crash, the failure of the object O is restricted to fail by crash, and this is difficult to achieve. This explains why adding graceful degradation to t-tolerance is not always possible.

An example To give an operational intuition of the previous impossibility result, this section considers a particular case, namely the construction of Fig. 15.10, which is gracefully degrading for the omission failure mode. As this construction is t-tolerant for omission failures, it is t-tolerant for the less severe crash failure mode. The following discussion shows that this *particular* construction is not gracefully degrading for the crash failure mode.

Let us consider a system of two processes p_i and p_j and an execution in which p_i invokes R_CONS.propose(0) while p_j invokes R_CONS.propose(1). Let us consider the following sequential scenario when p_i and p_j executes the code of the algorithm R_CONS.propose().

1. p_i executes $CONS[1]$.propose(0) and decides 0 (line 3).
2. p_j executes $CONS[1]$.propose(1) and decides 0 (line 3).
3. Then, p_j executes $CONS[k]$.propose(0) for $\le k \le 2t+1$ and decides 0 at each of these base consensus instances (lines 2, 3 and 8).

   It follows that, after p_j has exited the **for** loop, we have $dec_j[1..(2t+1)] = [0, 0, \ldots, 0]$ and consequently p_j decides 0 (lines 9–10).

4. After p_j has decided 0, all the base consensus objects crash.
5. Then, p_i continues its execution and executes the **for** loop for $2 \leq k \leq 2t + 1$. For any of these values of k, $CONS[k]$.propose() returns $\perp$ to p_i and we have $dec_j[1..(2t + 1)] = [0, \perp, \ldots, \perp]$ when p_i exits the loop. It follows that p_i returns the value $\perp$ at line 10.

In this execution more than t base objects fail and the resulting object *R_CONS* fails by omission (with respect to p_i). We show that this execution is not linearizable when considering the crash failure mode.

To be correct, the previous execution has to satisfy the sequential specification of consensus in the presence of crash failures. We show this is not the case. There are two cases according to which *R_CONS*.propose() is linearized first:

- Case 1. The invocation *R_CONS*.propose(0) by p_i (which returns the value $\perp$) is linearized before the invocation *R_CONS*.propose(1) by p_j (which returns the value 0).

  This case is not possible for the crash failure mode (it does not respect the property "once $\perp$, forever $\perp$" of the crash failure mode).
- Case 2. The invocation *R_CONS*.propose(1) by p_j (which returns the value 0) is linearized before the invocation *R_CONS*.propose(0) by p_i (which returns the value $\perp$).

  This case is not possible either because, according to the sequential specification of consensus, being the first invocation, *R_CONS*.propose(1) has to return the value 1.

It follows that no linearization of the invocations *R_CONS*.propose(0) by p_i and *R_CONS*.propose(1) by p_j is correct for the crash failure mode. Consequently, this particular construction, which is t-tolerant for responsive crash failures, is not gracefully degrading for this failure mode. Actually, what was proved by Jayanti, Chandra, and Toueg is the impossibility of such a construction.

15.5 Summary

Wait-freedom is a process-oriented liveness condition: it requires that the failure of a process does not prevent the progress of the other processes. Wait-free constructions described in the previous chapters have implicitly assumed that the objects shared by the processes are failure-free.

This chapter has considered the case where the base objects used in a construction are not necessarily reliable. It has focused on the self-implementation of two of the most important objects when the base objects from which they are built may fail, namely read/write registers and consensus objects. Self-implementation means that the base objects are of the same type as the object that is built. Three failure modes have been investigated: crash, omission, and arbitrary failures.

An object is t-tolerant with respect to a failure mode if it works correctly despite the failure of up to t base objects from which it is built. Wait-free self-implementations for read/write registers and consensus objects have been presented that are t-tolerant for several responsive failure modes.

Then, the chapter has introduced the notion of a gracefully degrading t-tolerant implementation and has presented such a consensus implementation for the omission failure mode. An implementation is gracefully degrading if, when more than t base objects fail, the constructed object does not fail more severely than the base objects from which it is built.

15.6 Bibliographic Notes

- The notions of t-tolerance and graceful degradation presented in this chapter are due to P. Jayanti, T.D. Chandra, and S. Toueg [169].
- The constructions presented in Figs. 15.8 and 15.9 are from [169]. This paper presents a suite of t-tolerant constructions (some being gracefully degrading) and impossibility results.
- The gracefully degrading consensus construction for responsive omission failure presented in Fig. 15.10 is due to P. Jayanti, T.D. Chandra, and S. Toueg [170].

  It is shown in [170] that this construction (which uses $2t+1$ base consensus objects) is space-optimal. As the construction described in Fig. 15.8 (which uses $t + 1$ base consensus objects) is t-tolerant (but not gracefully degrading) with respect to the omission failure mode, it follows that graceful degradation for consensus and omission failures has a price: the number of base objects increases by t.
- The space-optimal t-tolerant self-implementation of an SWSR atomic register described in Fig. 15.3 is due to R. Guerraoui and M. Raynal [131].
- The notion of *memory failure* is investigated in [8]. Such a failure corresponds to a faulty write.

15.7 Exercises and Problems

1. Design a wait-free t-tolerant self-implementation of an atomic SWMR register from base atomic registers prone to responsive crash failures.
2. Prove Theorem 65 (there is no wait-free t-tolerant self-implementation of a consensus object from crash-prone non-responsive consensus objects).
3. Design a t-tolerant wait-free self-implementation of an SWSR safe register where the base safe registers are prone to non-responsive arbitrary failures. (Hint: the solution uses $m = 5t + 1$ base safe registers.)

   Solution in [169].

4. Design a gracefully degrading 1-tolerant wait-free self-implementation of an SWSR safe register where the base safe registers are prone to responsive omission failures. (Hint: the solution uses $m = 4$ base safe registers.)

   Solution in [169].

5. Considering the responsive crash failure mode, design a t-tolerant wait-free self-implementation I of a compare&swap object. Then, show that I cannot be gracefully degrading.

6. Considering the responsive omission failure mode, design a t-tolerant wait-free self-implementation I of a compare & swap object that is gracefully degrading.

   Solution in [234].

7. Considering the responsive omission failure mode, design a t-tolerant wait-free self-implementation I of a read/write register that is gracefully degrading.

8. When considering the responsive omission failure mode, what can be concluded on universal constructions from the existence of the gracefully degrading constructions obtained in Exercises 6 and 7?

Chapter 16
Consensus Numbers and the Consensus Hierarchy

An object type characterizes the possible behaviors of a set of objects (namely, the objects of that type). As an example the type *consensus* defines the behavior of all consensus objects. Similarly, the type *atomic register* defines the behavior of all atomic registers. This chapter considers concurrent object types defined by a sequential specification on a finite set of operations.

We have seen in Chap. 14 that an object type T is *universal* if, together with atomic registers, objects of type T allow for the wait-free construction of any type T' (defined by a sequential specification on total operations). As consensus objects are universal, a natural and fundamental question that comes to mind is the following:

Which object types allow us to wait-free implement a consensus object?

As an example, do atomic registers alone or atomic registers plus queues allow consensus objects to be wait-free implemented? This chapter addresses this question. To that end it first introduces the notion of a *consensus number* that can be associated with an atomic object type. (In the following, when there is no confusion, we sometimes use equivalently the words "object" and "object type".) It then shows that the consensus number notion allows for the ranking of atomic objects with respect to their synchronization power when one is interested in wait-free object implementations.

Keywords Atomic object · Bivalent configuration · Consensus object · Consensus number · Consensus hierarchy · Indistinguishability · Monovalent configuration · Valence · Wait-freedom

16.1 The Consensus Number Notion

Consensus number The *consensus number* associated with an object type T is the largest number n such that it is possible to wait-free implement a consensus object from atomic read/write registers and objects of type T in a system of n processes.

If there is no largest n, the consensus number is said to be infinite. The consensus number associated with an object type T is denoted $CN(T)$. Let us observe that $CN(T) \geq 1$.

The *consensus number* notion is central to capture the power of object types when one is interested in wait-free implementations. This is formalized by the following theorem.

Theorem 68 *Let X and Y be two atomic object types such that $CN(X) = m$, $CN(Y) = n$, and $m < n$. There is no wait-free implementation of an object of type Y from objects of type X and read/write registers in a system of n processes.*

Proof As $CN(Y) = n$, objects of type Y (with atomic registers) allow building a wait-free implementation of a consensus object in a system of n processes. Similarly, we conclude from $CN(X) = m$ that a consensus object can be wait-free implemented from objects of type X (and atomic registers) in a system of m processes.

Let us assume (by contradiction) that it is possible to wait-free implement an object of type Y from atomic registers and objects of type X in a system of n processes. As $CN(Y) = n$, it follows that objects of type X can wait-free implement a consensus object in a system of n processes and we consequently have $CN(X) \geq n$. But, this contradicts the fact that $CN(X) = m < n$. $\square$

It could be the case that the consensus number of all concurrent atomic objects is $+\infty$, making the previous theorem useless. This chapter shows that this is not the case. It follows that the consensus number notion allows for the definition of a hierarchy that captures the power of concurrent objects when one wants to build wait-free object implementations. The rank of each object (type) in this hierarchy is its consensus number.

16.2 Fundamentals

16.2.1 Schedule, Configuration, and Valence

This section defines notions (schedule, configuration, and valence) which are central to prove the impossibility of wait-free implementing a consensus object from "too weak" object types in a system of n processes.

Reminder Let us consider an execution made up of sequential processes that invoke operations on atomic objects of types $T_1, \ldots, T_x$. These objects are called "base objects" (equivalently, the types $T_1, \ldots, T_x$ are called "base types"). We have seen in Chap. 4 (Theorem 14) that, as each base object is atomic, an execution at the operation level can be modeled by an atomic history (linearization) $\widehat{S}$ on the operation invocations issued by the processes. This means that (a) $\widehat{S}$ is a sequential history that includes all the operation invocations issued by the processes (except possibly the last operation invocation of a process if that process crashes), (2) $\widehat{S}$ is legal, and (3) $\widehat{S}$ respects the real-time occurrence order on the operation invocations.

Schedules and configurations A *schedule* is a sequence of operation invocations issued by processes. Sometimes the invocation of an operation is represented in a schedule only by the name of the process that issues that operation.

A *configuration* C is a global state of the system execution at a given point in time. It includes the value of each base object plus the local state of each process. The configuration $p(C)$ denotes the configuration obtained from C by applying an operation issued by the process p. More generally, given a schedule S and a configuration C, $S(C)$ denotes the configuration obtained by applying to C the sequence of operation invocations defining S.

Valence The *valence* notion is a fundamental concept to prove consensus impossibility results. Let us consider a consensus object such that only the values 0 and 1 can be proposed by the processes. As we have seen in Sect. 6.3.1, such an object is called a *binary consensus* object. Let us assume that there is an algorithm A implementing such a consensus object from base type objects. Let C be a configuration attained during an execution of the algorithm A.

The configuration C is *v-valent* if, no matter how the schedule defined by A is applied to C, it always leads to v as the decided value; v is the valence of that configuration. If $v = 0$ ($v = 1$) C is said to be 0-valent (1-valent). A 0-valent or 1-valent configuration is said to be *monovalent*. A configuration that is not monovalent is said to be *bivalent*.

Let us consider the prefix of an execution of a binary consensus algorithm A that attains configuration C.

While a monovalent configuration states that the decided value is determined (whether processes are aware of it or not), the decided value is not yet determined in a bivalent configuration. If C is monovalent, the value v decided by A could be deterministically computed from C by an external observer (this does not mean that processes are aware that the value that will be decided is determined, because they know neither C nor the fact that it is monovalent). Differently, if C is bivalent, the value that will be decided cannot be computed from C by an external observer. That value depends on the way the execution will progress from C (due to asynchrony and crashes).

16.2.2 Bivalent Initial Configuration

The next theorem shows that, for any wait-free consensus algorithm A, there is at least one initial bivalent configuration C, i.e., a configuration in which the decided value is not predetermined: there are several proposed values that can still be decided (for each of these values v, there is a schedule generated by the algorithm A that, starting from that configuration, decides v).

This means that, while the decided value is only determined from the inputs when the initial configuration is univalent, this is not true for all configurations, as there is at least one initial bivalent configuration: the value decided by a wait-free consensus algorithm cannot always be deterministically determined from the inputs. As indicated previously, it may depend on the execution of the algorithm A itself: according to the set of proposed values, a value can be decided in some runs while the other value can be decided in other executions.

Theorem 69 *Let us assume that there is an algorithm A that wait-free implements a binary consensus object in a system of n processes. There is then a bivalent initial configuration.*

Proof Let C_0 be the initial configuration in which all the processes propose 0 to the consensus object, and C_i, $1 \le i \le n$, the initial configuration in which the processes from p_1 to p_i propose the value 1, while all the other processes propose 0. So, all the processes propose 1 in C_n. These configurations constitute a sequence in which any two adjacent configurations C_{i-1} and C_i, $1 \le i \le n$, differ only in the value proposed by the process p_i: it proposes the value 0 in C_{i-1} and the value 1 in C_i. Moreover, it follows from the validity property of the consensus algorithm A that C_0 is 0-valent while C_n is 1-valent.

Let us assume that all the previous configurations are univalent. It follows that, in the previous sequence, there is (at least) one pair of consecutive configurations, say C_{i-1} and C_i, such that C_{i-1} is 0-valent and C_i is 1-valent. We show a contradiction.

Assuming that no process crashes, let us consider an execution history $\widehat{H}$ of the algorithm A that starts from the configuration C_{i-1}, in which the process p_i invokes no base operation for an arbitrarily long period (the end of that period will be defined below). As the algorithm is wait-free, the processes decide after a finite number of invocations of base operations. The sequence of operations that starts at the very beginning of the history and ends when all the processes have decided (except p_i, which has not yet started invoking base operations) defines a schedule S. (See the upper part of Fig. 16.1. Within the vector containing the values proposed by each process, the value proposed by p_i is placed inside a box.) Then, after S terminates, p_i starts executing and eventually decides. As C_{i-1} is 0-valent, $S(C_{i-1})$ is also 0-valent.

Let us observe (lower part of Fig. 16.1) that the same schedule S can be produced by the algorithm A from the configuration C_i. This is because (1) the configurations

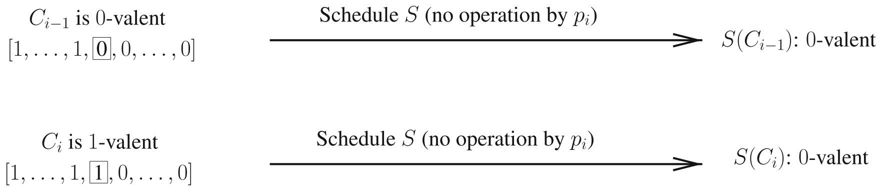

C_{i-1} is 0-valent
$[1,\ldots,1,\boxed{0},0,\ldots,0]$
Schedule S (no operation by p_i)
$S(C_{i-1})$: 0-valent
C_i is 1-valent
$[1,\ldots,1,\boxed{1},0,\ldots,0]$
Schedule S (no operation by p_i)
$S(C_i)$: 0-valent

Fig. 16.1 Existence of a bivalent initial configuration

C_{i-1} and C_i differ only in the value proposed by p_i and (2) as p_i invokes no base operations in S, that schedule cannot depend on the value proposed by p_i. As $S(C_{i-1})$ is 0-valent, it follows that the configuration $S(C_i)$ is also 0-valent. But, as also C_i is 1-valent, we conclude that $S(C_i)$ is 1-valent, a contradiction. □

Crash versus asynchrony The previous proof is based on (1) the assumption stating that the consensus algorithm A is wait-free (intuitively, the progress of a process does not depend on the "speed" of the other processes) and (2) asynchrony (a process progresses at its "own speed"). This allows the proof to play with process speeds and consider a schedule (part of an execution history) in which a process p_i does not execute operations. We could have instead considered that p_i has initially crashed (i.e., p_i crashes before executing any operation). During the schedule S, the wait-free consensus algorithm A (the existence of which is a theorem assumption) has no way to distinguish between these two cases (has p_i initially crashed or is it only very slow?). This shows that, for some problems, asynchrony and process crashes are two facets of the same "uncertainty" with which wait-free algorithms have to cope.

16.3 The Weak Wait-Free Power of Atomic Registers

We have seen in Part III that the computability power of atomic read/write registers is sufficient to build wait-free implementations of concurrent objects such as splitters, weak counters, store-collect objects, and snapshot objects. As atomic read/write registers are very basic objects, an important question from a computability point of view, is then: are atomic read/write registers powerful enough to build wait-free implementations of other concurrent objects such as queues, stacks, etc.

This section shows that the answer to this question is "no". More precisely, it shows that MWMR atomic registers are not powerful enough to wait-free implement a consensus object in a system of two processes. This means that the consensus number of an atomic read/write register is 1; i.e., read/write registers allow a consensus object to be wait-free implemented only in a system made up of a single process. Stated another way, atomic read/write registers have the "poorest" computability power when one is interested in wait-free implementations of atomic objects in asynchronous systems prone to any number of process crashes.

16.3.1 The Consensus Number of Atomic Read/Write Registers Is 1

To show that there is no algorithm that wait-free implements a consensus object in a system of two processes p and q, the proof assumes such an algorithm and derives a contradiction. The central concept in this proof is the notion of valence previously introduced.

Theorem 70 *There is no algorithm A that wait-free implements a consensus object from atomic read/write registers in a set of two processes (i.e., the consensus number of atomic registers is* 1.*)*

Proof Let us assume (by contradiction) that there is an algorithm A that uses only atomic read/write registers and builds a wait-free implementation of a consensus object in a system of two processes. Due to Theorem 69, there is an initial bivalent configuration. The proof of the theorem consists in showing that, starting from such a bivalent configuration C, there is always an arbitrarily long schedule S produced by A that, starting from C, makes the execution progress to another bivalent configuration $S(C)$. Then, by repeating this argument, it follows that A has an execution in which no process ever decides, which proves the theorem.

Given a configuration D, let us recall that $p(D)$ is the configuration obtained by applying the next operation invoked by process p—as defined by the algorithm A—to the configuration D. Let us also recall that the operations that p or q can issue are reading or writing of base atomic registers.

Let us observe that, starting the algorithm from the bivalent configuration C, there is a maximal schedule S such that $D = S(C)$ is bivalent. "Maximal" means that both the configuration $p(D)$ and the configuration $q(D)$ are monovalent and have different valence (otherwise, D would not be bivalent). Without loss of generality, let us consider that $p(D)$ is 0-valent while $q(D)$ is 1-valent.

The operation that leads from D to $p(D)$ is an invocation by p of $R1.\mathsf{read}()$ or $R1.\mathsf{write}()$, where $R1$ is a base atomic register. Similarly, the operation that leads from D to $q(D)$ is an invocation by q of $R2.\mathsf{read}()$ or $R2.\mathsf{write}()$, where $R2$ is a base atomic register. The proof consists in a case analysis:

1. $R1$ and $R2$ are distinct registers (Fig. 16.2).
   In that case, as $R1$ and $R2$ are distinct registers, whatever are the (read or write) operations $R1.\mathsf{op}_1()$ and $R2.\mathsf{op}_2()$ issued by p and q on the atomic registers $R1$ and $R2$, the configurations $p(q(D))$ and $q(p(D))$ are the same configuration, i.e., $p(q(D)) \equiv q(p(D))$, where the symbol "$\equiv$" means that the configurations

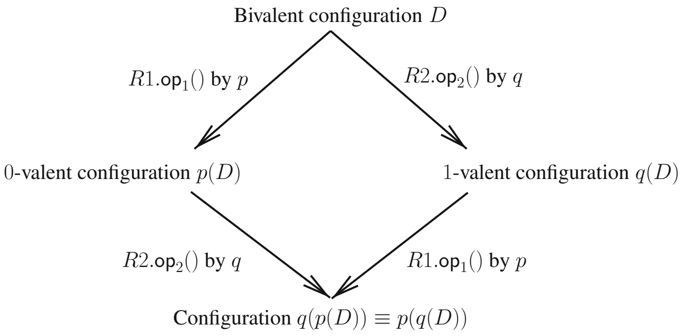

Bivalent configuration D
$R1.\mathsf{op}_1()$ by p
$R2.\mathsf{op}_2()$ by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$
$R2.\mathsf{op}_2()$ by q
$R1.\mathsf{op}_1()$ by p
Configuration $q(p(D)) \equiv p(q(D))$

Fig. 16.2 Read/write invocations on distinct registers

$p(q(D))$ and $q(p(D))$ are the very same configuration (each process is in the same local state and each shared register has the same value in both configurations).
As $q(D)$ is 1-valent, it follows that $p(q(D))$ is also 1-valent. Similarly, as $p(D)$ is 0-valent, it follows that $q(p(D))$ is also 0-valent. This is a contradiction, as the configuration $p(q(D)) \equiv q(p(D))$ cannot be both 0-valent and 1-valent.

2. $R1$ and $R2$ are the same register R.

   - Both p and q read R.
     As a read operation on an atomic register does modify its value, this case is the same as the previous one (where p and q access distinct registers).
   - p invokes R.read(), while q invokes R.write() (Fig. 16.3).
     (Let us notice that the case where q reads R while p writes R is similar.) Let $Read_p$ be the read operation issued by p on R, and $Write_q$ be the write operation issued by q on R. As $Read_p(D)$ is 0-valent, so is $Write_q(Read_p(D))$. Moreover, $Write_q(D)$ is 1-valent.

     The configurations D and $Read_p(D)$ differ only in the local state of p (it has read R in the configuration $Read_p(D)$, while it has not in D). These two configurations cannot be distinguished by q. Let us consider the following two executions:

     – After the configuration D was attained by the algorithm A, p stops executing for an arbitrarily long period, and during that period only q executes base read/write operations (on base atomic read/write registers). As by assumption the algorithm A is wait-free, there is a finite sequence of read/write invocations on atomic registers issued by q at the end of which q

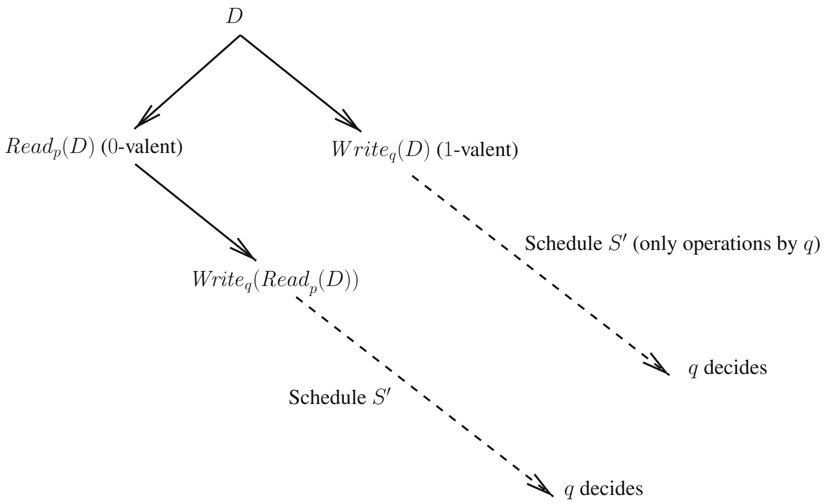


Fig. 16.3 Read and write invocations on the same register

decides. Let S' be the schedule made up of these invocations. As $Write_q(D)$ is 1-valent, it decides 1. (Thereafter, p wakes up and executes read/write invocations as specified in the algorithm A. Alternatively, p could crash after the configuration D has been attained.)

– Similarly, after the configuration $Read_p(D)$ was attained by the algorithm A, p stops executing for an arbitrarily long period. The same schedule S' (defined in the previous item) can be issued by q after the configuration $Read_p(D)$. This is because, as p issues no read/write invocations on base atomic registers, q cannot distinguish D from $Read_p(D)$. It follows that q decides at the end of that schedule, and, as $Write_q(Read_p(D))$ is 0-valent, q decides 0.

While executing the schedule S', q cannot know which (D or $Read_p(D)$) was the configuration when it started executing S' (this is because these configurations differ only in a read of R by p). As the schedule S' is deterministic (it is composed only of read and write invocations issued by q on base atomic registers), q must decide the same value, whatever the configuration at the beginning of S'. This is a contradiction, as it decides 0 in the first case and 1 in the second case.

- Both p and q invoke R.write().
Let $Write_p$ and $Write_q$ be the write operations issued by p and q on R, respectively. By assumption the configurations $Write_p(D)$ and $Write_q(D)$ are 0-valent and 1-valent, respectively.

The configurations $Write_q(Write_p(D))$ and $Write_q(D)$ cannot be distinguished by q: the write of R by p in the configuration D that produces the configuration $Write_p(D)$ is overwritten by q when it produces the configuration $Write_q(Write_p(D))$.

The reasoning is then the same as the previous one. It follows that, if q executes alone from D until it decides, it decides 1 after executing a schedule S''. The same schedule from the configuration $Write_p(D)$ leads it to decide 0. But, as q cannot distinguish D from $Write_p(D)$, and S'' is deterministic, it follows that it has to decide the same value in both executions, a contradiction as it decides 0 in the first case and 1 in the second case. (Let us observe that Fig. 16.3 is still valid. We have only to replace $Read_p(D)$ and S' by $Write_p(D)$ and S'', respectively.) □

16.3.2 The Wait-Free Limit of Atomic Registers

Theorem 71 *It is impossible to construct a wait-free implementation of an object with consensus number greater than* 1 *from atomic registers.*

Proof The proof is an immediate consequence of Theorem 70 (the consensus number of atomic registers is 1) and Theorem 68 (if $CN(X) < CN(Y)$, X does not allow for the construction of a wait-free implementation of Y in a system of more than $|CN(X)|$ processes). □

The following corollary is an immediate consequence of the fact that both snapshot objects and renaming objects can be wait-free implemented from atomic read/write registers (see Chaps. 8 and 9).

Corollary 7 *The consensus number of a snapshot object is* 1. *The consensus number of an M-renaming object (where $M \geq 2p - 1$ for size-adaptive renaming when p processes participate and $M \geq 2n - 1$ for non-adaptive renaming) is* 1.

16.4 Objects Whose Consensus Number Is 2

As atomic registers are too weak to wait-free implement a consensus object for two processes, the question posed at the beginning of the chapter becomes: are there objects that allow for the construction of a wait-free implementation of consensus objects in systems of two or more processes.

This section first considers three base objects (test&set objects, queue, and swap objects) and shows that, together with shared registers, they can wait-free implement consensus in a set of two processes. These objects and their operations have been introduced in Sect. 2.2 (their definition is repeated in the following to make this chapter easier to read). This section then shows that these concurrent objects cannot wait-free implement a consensus object in a system of three or more processes. It follows that their consensus number is 2.

In this section, when considering two processes, they are denoted p_0 and p_1 (considering the process indexes 0 and 1 makes the presentation simpler). When considering three processes, they are denoted p, q, and r.

16.4.1 Consensus from Test&Set Objects

Test&set objects A test&set object *TS* is a one-shot atomic object that provides the processes with a single operation called test&set() (hence the name of the object). Such an object can be seen as maintaining an internal state variable X which is initialized to 1 and can contain only the values 0 or 1. The effect of an invocation of TS.test&set() can be described by the atomic execution of the following code:

$$prev \leftarrow X; \textbf{if } (prev = 1) \textbf{ then } X \leftarrow 0 \textbf{ end if}; \mathsf{return}(prev).$$

```
operation $C$.propose($v$) is
(1)   $REG[i] \leftarrow v$;
(2)   $aux \leftarrow TS$.test&set();
(3)   case $(aux = 1)$ then return$(REG[i])$
(4)         $(aux = 0)$ then return$(REG[1-i])$
(5)   end case
end operation.
```

Fig. 16.4 From test&set to consensus (code for p_i, $i \in \{0, 1\}$)

Hence, the test&set object in which we are interested is an atomic one-shot object whose first invocation returns 1 while all the other invocations return 0. The process that obtains the value 1 is called the *winner*. The other ones (or the other one in the case of two processes) are called *losers*.

From test&set to consensus for two processes The algorithm described in Fig. 16.4 builds a wait-free implementation of a consensus object for two processes (p_0 and p_1) from a test&set object *TS*. It uses two additional SWSR registers $REG[0]$ and $REG[1]$ (a process p_i can always keep a local copy of the atomic register it writes, so we do not count it as one of its readers). Moreover, as each register $REG[i]$ is always written by p_i before being read by p_{1-i}, they do not need to be atomic (or even safe). Let C be the consensus object that is built. The construction is made up of two parts:

- When process p_i invokes C.propose(v), it deposits the value v it proposes into *REG*[i] (line 1). This part consists for p_i in making public the value it proposes.
- Then p_i executes a control part to know which value has to be decided. To that aim, it uses the underlying test&set object *TS* (line 2). If it obtains the initial value of the test&set object (1), it is the winner and decides the value it has proposed (line 3). Otherwise p_i is the loser and accordingly it decides the value proposed by the winner p_{1-i} (line 4).

Let us remember that, as the test&set object is atomic, the winner is the process whose invocation *TS*.test&set() is the first that appears in the linearization order associated with the object *TS*.

Theorem 72 *The algorithm described in Fig. 16.4 is a wait-free construction, in a system of two processes, of a consensus object from a test&set object.*

Proof The algorithm is clearly wait-free, from which follows the consensus termination property.

Let p_i be the winner. When it executes line 2, the test&set object *TS* changes its value from 1 to 0, and then, as any other invocation finds $TS = 0$, the test&set object keeps forever the value 0. As p_i is the only process that obtains the value 1 from *TS*, it decides the value v it has just deposited in *REG*[i] (line 3).

Moreover, as the other process p_{1-i} obtains the value 0 from *TS*, it decides the value deposited *REG*[i] by the winner p_i (line 4). Let us observe that, due to the values that are returned to p_i and p_{1-i} by the test&set object, we conclude that the invocation of TS.test&set() by p_i is linearized before the one issued by p_{1-i}, from which it follows that $REG[i]$ was written before being read by p_{1-i}.

It follows that a single value is decided (consensus agreement property) and that value was proposed by a process (consensus validity property). Hence, the algorithm described in Fig. 16.4 is a wait-free implementation of a consensus object in a system of two processes. □

16.4.2 Consensus from Queue Objects

Concurrent queue objects These objects are encountered in a lot of multiprocess programs and have been used already in several chapters. Here we restrict our attention to a one-shot queue whose initial value is not the empty sequence. Let us remember that a queue is defined by two total operations with a sequential specification. The operation denoted enqueue() adds an item at the end of the queue, while the operation denoted dequeue() removes the item at the head of the queue and returns it to the invoking process. If the queue is empty, the default value $\perp$ is returned.

We consider here a one-shot queue (a process issues at most one operation invocation) and that queue has a non-empty initial value.

From a concurrent atomic queue to consensus A wait-free algorithm that constructs, in a system of two processes, a consensus object C from an atomic queue Q is described in Fig. 16.5. This algorithm is based on the same principles as the previous one, and its code is nearly the same. The only difference is in line 2, where a queue Q is used instead of a test&set object. The queue is initialized to the sequence of $\langle w, \ell \rangle$. The process that dequeues w (the value at the head of the queue) is the winner, and the process that dequeues ℓ is the loser. The value decided by the consensus object is the value proposed by the winner.

Theorem 73 *The algorithm described in Fig. 16.5 is a wait-free construction of a consensus object from an atomic queue object, in a system of two processes.*

Proof The proof is the same as the proof of Theorem 72. The only difference is the way the winner process is selected. Here, the winner is the process that dequeues the

```
operation $C$.propose($v$) is
(1)     $REG[i] \leftarrow v$;
(2)     $aux \leftarrow Q$.dequeue();
(3)     case $(aux = w)$ then return($REG[i]$)
(4)          $(aux = \ell)$ then return($REG[1-i]$)
(5)     end case
end operation.
```

Fig. 16.5 From an atomic concurrent queue to consensus (code for p_i, $i \in \{0, 1\}$)

```
operation $C$.propose($v$) is
(1)    $REG[i] \leftarrow v$;
(2)    $aux \leftarrow R$.swap($i$);
(3)    case ($aux = \perp$) then return($REG[i]$)
(4)         ($aux \neq \perp$) then return($REG[1-i]$)
(5)    end case
end operation.
```

Fig. 16.6 From a swap register to consensus (code for p_i, $i \in \{0, 1\}$)

value w which is initially at the head of the queue. As suggested by the text of the algorithm, the proof is then verbatim the same. □

16.4.3 Consensus from Swap Objects

Swap objects The swap object R considered here is a one-shot atomic register that can be accessed by an operation denoted R.swap(). This operation, which has an input parameter v, assigns it to R and returns the previous value of R. Its effect can be described by the atomic execution of the following statements:

$$prev \leftarrow R;\ R \leftarrow v;\ \mathsf{return}(prev).$$

From swap objects to consensus An algorithm that wait-free implements a consensus object from a swap object in a system of two processes is described in Fig. 16.6. The swap object R is initialized to $\perp$. The design principles of this construction are the same as the ones of the previous algorithms. The winner is the process that succeeds in depositing its index in R while obtaining the value $\perp$ from R. The proof of the algorithm is the same as the proof of the previous algorithms.

16.4.4 Other Objects for Wait-Free Consensus in a System of Two Processes

It is possible to build a wait-free implementation of a consensus object, in a system of two processes, from other objects such as a concurrent stack, a concurrent list, or a concurrent priority queue. When they do not provide total operations, the usual definition of these objects can easily be extended in order to have only total operations. As an example, the stack operation pop() can be extended by returning the default value $\perp$ when the stack is empty.

Other objects such as fetch&add objects allow for wait-free construction of a consensus object in a system of two processes.

Such an object has a single operation $\mathsf{fetch\&add}(x)$ whose input parameter x is a positive integer. Moreover, an invocation of $\mathsf{fetch\&add}(x)$ returns a value.

The internal state of a fetch&add object is kept in a read/write register X, and the effect of an invocation of $\mathsf{fetch\&add}(x)$ can be described by the atomic execution of the following sequence of statements:

$$prev \leftarrow X;\ X \leftarrow X + x;\ \mathsf{return}(prev).$$

The construction of a consensus object requires a single one-shot fetch&add object.

16.4.5 Power and Limit of the Previous Objects

The computational power of the previous objects allows for wait-free implementations of a consensus object in a system of two processes, hence the question: Do they allow for the wait-free implementation of a consensus object in a system of three or more processes? Surprisingly, the answer to this question is "no".

This section gives a proof that the consensus number of an atomic concurrent queue is exactly 2. (The proofs for the other objects presented in this section are similar.)

Theorem 74 *Atomic wait-free queues have consensus number 2.*

Proof This proof has the same structure as the proof of Theorem 70. Considering binary consensus, it assumes that there is an algorithm based on queues and atomic read/write registers that wait-free implements a consensus object in a system of three processes (denoted p, q, and r). As in Theorem 70 we show that, starting from an initial bivalent configuration C (due to Theorem 69, such a configuration does exist), there is an arbitrarily long schedule S produced by A that leads from C to another bivalent configuration $S(C)$. This shows that A has an execution in which no process ever decides, which proves the theorem by contradiction.

Starting the algorithm A in a bivalent configuration C, let S be a maximal schedule produced by A such that the configuration $D = S(C)$ is bivalent. As we have seen in the proof of Theorem 70, "maximal" means that the configurations $p(D)$, $q(D)$, and $r(D)$ are monovalent. Moreover, as D is bivalent, two of these configurations have different valences. Without loss of generality let us say that $p(D)$ is 0-valent and $q(D)$ is 1-valent. $r(D)$ is either 0-valent or 1-valent (the important point here is that $r(D)$ is not bivalent).

Let Op_p be the invocation of the operation issued by p that leads from D to $p(D)$, Op_q the invocation of the operation issued by q that leads from D to $q(D)$, and Op_r the operation issued by r that leads from D to $r(D)$. Each of Op_p, Op_q, and Op_r is a read or a write of an atomic register or an enqueue or a dequeue on an atomic queue.

Let us consider p and q (the processes that produce configurations with different valences), and let us consider that, from D, process r does not execute operations for an arbitrarily long period:

- If both Op_p and Op_q are invocations of operations on atomic registers, the proof follows from the proof of Theorem 70.
- If one of Op_p and Op_q is an invocation of an operation on an atomic register while the other is an invocation of an operation on an atomic queue, the reasoning used in item 1 of the proof of Theorem 70 applies. This reasoning, based on the argument depicted in Fig. 16.2, permits one to conclude that $p(q(D)) \equiv q(p(D))$, while one is 0-valent and the other is 1-valent.

It follows that the only case that remains to be investigated is when both Op_p and Op_q are operations on the same atomic queue Q. We proceed by a case analysis. There are three cases:

1. Both Op_p and Op_q are invocations of Q.dequeue().
   As $p(D)$ and $q(D)$ are 0-valent and 1-valent, respectively, the configuration $Op_q(Op_p(D))$ is 0-valent, while $Op_p(Op_q(D))$ is 1-valent. But these configurations cannot be distinguished by the process r: in both configurations, r has the same local state and each base object —atomic register or atomic queue— has the same value. So, starting from any of these configurations, let us consider a schedule S' in which only r invokes operations (as defined by the algorithm A) until it decides (the fact that neither p nor q invokes operations in this schedule is made possible by asynchrony). We have then (1) $S'(Op_q(Op_p(D))$ is 0-valent and (2) $S'(Op_p(Op_q(D))$ is 1-valent. But, as $Op_q(Op_p(D))$ and $Op_q(Op_p(D))$ cannot be distinguished by r, it has to decide the same value in both $S'(Op_q(Op_p(D))$ and $S'(Op_p(Op_q(D))$, which is a contradiction.
2. Op_p is an invocation of Q.dequeue(), while Op_q is the invocation Q.enqueue(a).
   (The case where Op_p is Q.enqueue(a) while Op_q is Q.dequeue() is the same.) There are two sub-cases according to the current state of the wait-free atomic queue Q:

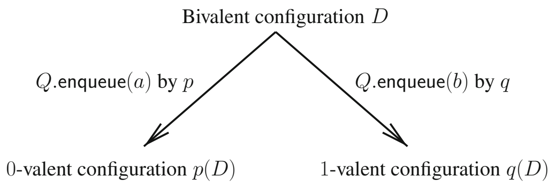

Bivalent configuration D
Q.enqueue(a) by p
Q.enqueue(b) by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$

Fig. 16.7 Q.enqueue() invocations by the processes p and q

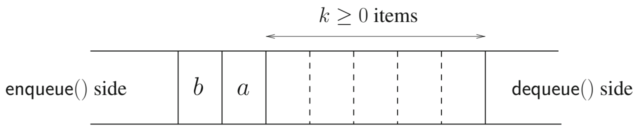


Fig. 16.8 State of the atomic queue Q in configuration $q(p(D))$

- Q is not empty. In this case, the configurations $Op_q(Op_p(D))$ and $Op_p(Op_q(D))$ are the same configuration: in both, each object has the same state and each process is in the same local state. This is a contradiction because $Op_q(Op_p(D))$ is 0-valent while $Op_p(Op_q(D))$ is 1-valent.
- Q is empty. In this case, r cannot distinguish the configuration $Op_p(D)$ and $Op_p(Op_q(D))$. The same reasoning as that of case 1 above shows a contradiction (the same schedule S' starting from any of these configurations and involving only operation invocations issued by r has to decide both 0 and 1).

3. Op_p is the invocation of Q.enqueue(a) and Op_q is the invocation of Q.enqueue(b). (This case is described in Fig. 16.7.)

   Let k be the number of items in the queue Q in the configuration D. This means that $p(D)$ contains $k+1$ items, and $q(p(D))$ (or $p(q(D))$) contains $k+2$ items (see Fig. 16.8).

   As the algorithm A is wait-free and $p(D)$ is 0-valent, there is a schedule S_p starting from the configuration $q(p(D))$ that includes only operation invocations issued by p (these invocations are on atomic registers and possibly other atomic queues) and that ends with p deciding 0.

   Claim C1. The schedule S_p contains an invocation by p of Q.dequeue() that dequeues the $(k+1)$th element of Q.
   Proof of the claim C1. Assume by contradiction that p issues at most k invocations of Q.dequeue() in S_p (and so it never dequeues the value a it has enqueued). In that case (see Fig. 16.9), if we apply the schedule pS_p (i.e., the invocation of Q.enqueue(a) by p followed by the schedule S_p) to the configuration $q(D)$, we obtain the configuration $S_p(p(q(D)))$ in which p decides 0. It decides 0 for the following reason: as it dequeues at most k items from Q, p cannot distinguish the configurations $p(q(D))$ and $q(p(D))$. This is because these two configurations differ only in the state of the queue Q: its two last items in $q(p(D))$ are a followed by b while they are b followed by a in $p(q(D))$, but, as we have just seen, p decides 0 in $S_p(q(p(D)))$ without having dequeued a or b (let us remember that, due to the contradiction assumption, S_p contains at most k dequeues of Q). But this contradicts the fact that, as $q(D)$ is 1-valent, p should have decided 1. Hence, S_p contains an invocation Q.dequeue() that dequeues the $(k+1)$th element of Q. End of proof of the claim C1.

   It follows from this claim that S_p contains at least $k+1$ invocations of Q.dequeue() issued by p. Let S'_p be the longest prefix of S_p that does not contain the $(k+1)$th dequeue invocation on Q by p:

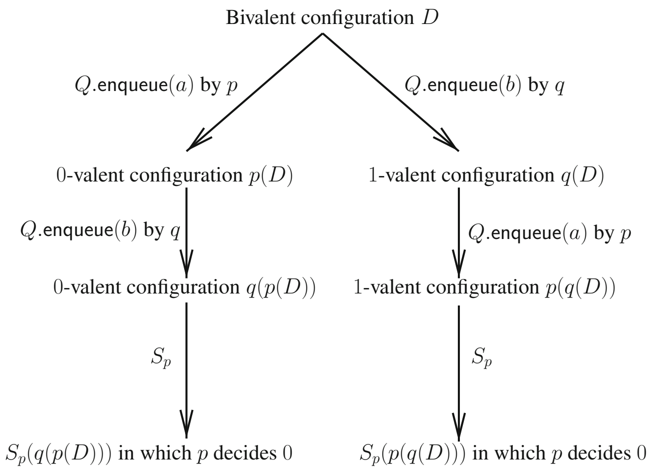

Bivalent configuration D
Q.enqueue(a) by p
Q.enqueue(b) by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$
Q.enqueue(b) by q
Q.enqueue(a) by p
0-valent configuration $q(p(D))$
1-valent configuration $p(q(D))$
S_p
S_p
$S_p(q(p(D)))$ in which p decides 0
$S_p(p(q(D)))$ in which p decides 0

Fig. 16.9 Assuming that S_p contains at most k invocations of Q.dequeue()

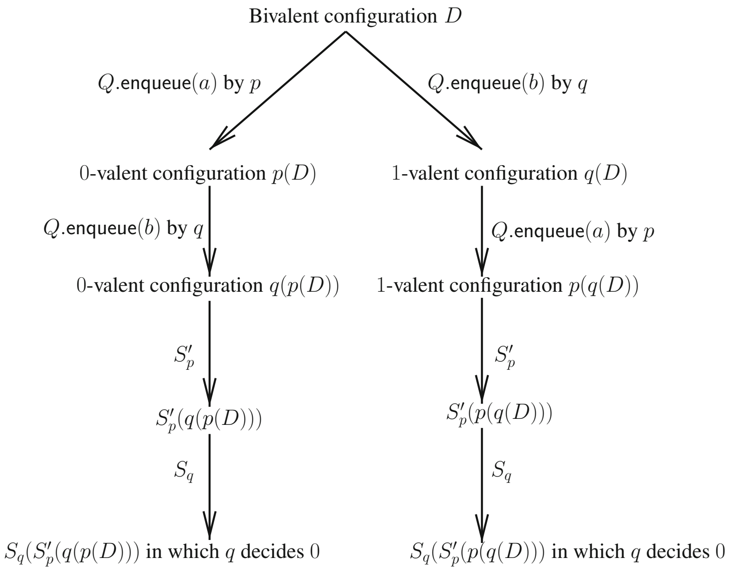

Bivalent configuration D
Q.enqueue(a) by p
Q.enqueue(b) by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$
Q.enqueue(b) by q
Q.enqueue(a) by p
0-valent configuration $q(p(D))$
1-valent configuration $p(q(D))$
S'_p
S'_p
$S'_p(q(p(D)))$
$S'_p(p(q(D)))$
S_q
S_q
$S_q(S'_p(q(p(D)))$ in which q decides 0
$S_q(S'_p(p(q(D)))$ in which q decides 0

Fig. 16.10 Assuming that S_q does not contain invocations of Q.dequeue()

$$S_p = \mathsf{inv}_1, \mathsf{inv}_2, \ldots, \mathsf{inv}_\mathsf{x}, (k+1)\text{th invocation of } Q.\mathsf{dequeue}(), \ldots, \mathsf{inv}_\mathsf{y}, \ldots$$
$$S'_p = \mathsf{inv}_1, \mathsf{inv}_2, \ldots, \mathsf{inv}_\mathsf{x}.$$

As the algorithm A is wait-free and $p(D)$ is 0-valent, there is a schedule S_q, starting from the configuration $S'_p(q(p(D)))$ and involving only operations from q, that ends with q deciding 0 (left side of Fig. 16.10).

Claim C2. The schedule S_q contains an invocation of Q.dequeue() by q.
Proof of the claim C2. Similarly to the above discussion, let us assume by contradiction that q never dequeues an item from Q in S_q. In this case, if we apply the schedule S_q to the configuration $S'_p(p(q(D)))$, q decides 0 (right side of Fig. 16.10). This is because the only difference between the configurations $S'_p(p(q(D)))$ and $S'_p(q(p(D)))$ is in the state of the queue Q. But, due to the contradiction assumption, q never dequeues an item from Q when it executes the schedule S_q, and consequently, it cannot distinguish $S'_p(p(q(D)))$ from $S'_p(q(p(D)))$. It follows that it has to decide 0 when it executes the schedule S_q from $S'_p(p(q(D)))$. This contradicts the fact that $q(D)$ is 1-valent and proves that S_q has to contain at least one invocation of Q.dequeue(). End of proof of the claim C2.

Let S'_q be the longest prefix of S_q that does not contain dequeue invocations issued by q in S_q.

$$S_q = \mathsf{inv_a}, \mathsf{inv_b}, \ldots, \mathsf{inv_i}, \text{ first invocation of } Q.\mathsf{dequeue}(), \ldots, \mathsf{inv_j}, \ldots$$
$$S'_q = \mathsf{inv_a}, \mathsf{inv_b}, \ldots, \mathsf{inv_i}.$$

We now define two schedules that start from D, lead to different decisions, and cannot be distinguished by the third process r (Fig. 16.11).

- The first schedule is defined by the following sequence of operation invocations:
  - p executes Q.enqueue(a) (producing $p(D)$),
  - q executes Q.enqueue(b) (producing $q(p(D))$),
  - p executes the operations in S'_p, then executes Q.dequeue() and obtains a,
  - q executes the operation invocations in S'_q, then executes Q.dequeue() and obtains b.

  This schedule leads from the configuration D to the configuration denoted D_0. As D_0 is reachable from $p(D)$, it is 0-valent.

- The second schedule is defined by the following sequence of operation invocations:
  - q executes Q.enqueue(b) (producing $q(D)$),
  - p executes Q.enqueue(a) (producing $p(q(D))$),
  - p executes the operation invocations in S'_p, then executes Q.dequeue() and obtains b,
  - q executes the operation invocations in S'_q, then executes Q.dequeue() and obtains a.

  This schedule leads from the configuration D to the configuration denoted D_1. As D_1 is reachable from $q(D)$, it is 1-valent.

Let us now consider the third process r (that has not invoked operations since configuration D).

All the objects have the same state in the configurations D_0 and D_1. Moreover, process r has also the same local state in both configurations. It follows that D_0

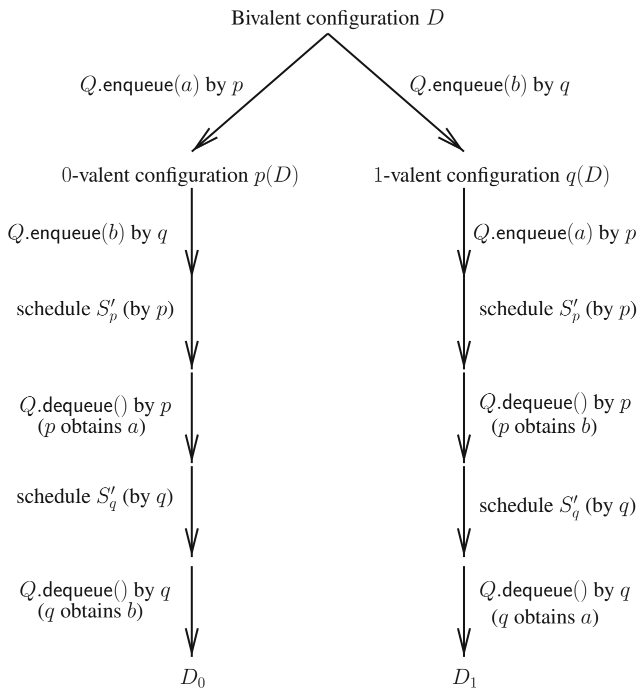

Bivalent configuration D
Q.enqueue(a) by p
Q.enqueue(b) by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$
Q.enqueue(b) by q
Q.enqueue(a) by p
schedule S'_p (by p)
schedule S'_p (by p)
Q.dequeue() by p (p obtains a)
Q.dequeue() by p (p obtains b)
schedule S'_q (by q)
schedule S'_q (by q)
Q.dequeue() by q (q obtains b)
Q.dequeue() by q (q obtains a)
D_0
D_1

Fig. 16.11 From the configuration D to the configuration D_0 or D_1

and D_1 cannot be distinguished by r. Consequently, as the algorithm A is wait-free and D_0 is 0-valent, there is a schedule S_r, starting from the configuration D_0 and involving only operation invocations issued by r, in which r decides 0. As r cannot distinguish D_0 and D_1, the very same schedule can be applied from D_1, at the end of which r decides 0. This is a contradiction, as D_1 is 1-valent. □

The following corollary is an immediate consequence of Theorems 71 and 74.

Corollary 8 *Wait-free atomic objects such as queues, stacks, lists, priority queues, test&set objects, swap objects and fetch&add objects cannot be wait-free implemented from atomic read/write registers.*

16.5 Objects Whose Consensus Number Is $+\infty$

This section shows that there are atomic objects whose consensus number is $+\infty$. Hence, it is possible to wait-free implement a consensus object in a system with read/write registers and such objects whatever the number n of processes, from which it follows that it is possible to wait-free implement in these systems any object defined by a sequential specification on total operations. Three such objects are presented here: compare&swap objects, memory to memory swap objects, and augmented queues.

16.5.1 Consensus from Compare&Swap Objects

As seen in previous chapters, a compare&swap object *CS* is an atomic object that can be accessed by a single operation that is denoted compare&swap(). This operation, which returns a value, has two input parameters (two values called old and new). To make the presentation easier we consider here that an invocation of compare&swap() returns the previous value of the compare&swap instead of a Boolean value. Its effect can be described by the atomic execution of the following sequence of statements (where X is an internal object where the current value of the compare&swap object is saved).

```
operation compare&swap($old, new$) is
    $prev \leftarrow X$;
    if $(X = old)$ then $X \leftarrow new$ end if;
    return($prev$)
end operation.
```

From compare&swap objects to consensus The algorithm described in Fig. 16.12 is a wait-free construction of a consensus object C from a compare&swap object in a system of n processes, for any value of n. The base compare&swap object, denoted *CS*, is initialized to $\perp$, a default value that cannot be proposed by the processes to the consensus object. When a process proposes a value v to the consensus object, it first invokes CS.compare&swap($\perp, v$) (line 1). If it obtains $\perp$, it decides the value it proposes (line 2). Otherwise, it decides the value returned from the compare&swap object (line 3). In the first case, the invoking process is the *winner*, while in the second case it is a *loser*.

Theorem 75 *Compare&swap objects have infinite consensus number.*

Proof The algorithm described in Fig. 16.12 is clearly wait-free. As the base compare&swap object *CS* is atomic, there is a first process that executes *CS*.

```
operation $C$.propose($v$) is
(1)   $aux \leftarrow CS$.compare&swap($\perp, v$);
(2)   case $aux = \perp$ then return($v$)
(3)        $aux \neq \perp$ then return($aux$)
(4)   end case
end operation.
```

Fig. 16.12 From compare&swap to consensus

compare&swap() (as usual, "first" is defined according to the linearization order of all the invocations CS.compare&swap()). This process is the winner. According to the specification of the compare&swap() operation, the winner has deposited v (the value it proposes to the consensus object) in CS. As the input parameter *old* of any invocation of the compare&swap() operation is $\perp$, it follows that all future invocations of the operation compare&swap() will return the single value deposited in CS, namely the value v deposited by the winner. It follows that all the processes that propose a value and do not crash decide the value of the winner. The algorithm is trivially independent of the number of processes that invoke CS.compare&swap(). It follows that the algorithm wait-free implements a consensus object for any number of processes. □

Remark on the number and the indexes of processes The previous consensus construction based on acompare&swap object uses neither the process indexes nor the number n of processes. It consequently works in anonymous systems with infinitely many processes.

16.5.2 Consensus from Mem-to-Mem-Swap Objects

Mem-to-mem-swap objects A mem-to-mem-swap object is an atomic register that provides the processes with two operations: the classical read operation plus the operation mem_to_mem_swap() that is on two mem-to-mem-swap registers. The invocation of $R1$.mem_to_mem_swap($R2$) atomically exchanges the content of the registers $R1$ and $R2$.

From mem-to-mem-swap objects to consensus The algorithm described in Fig. 16.13 is a wait-free construction of a consensus object from base atomic registers and mem-to-mem-swap objects, for any number n of processes.

```
operation $C$.propose($v$) is
(1)    $REG[i] \leftarrow v$;
(2)    $A[i]$.mem_to_mem_swap($R$);
(3)    for $j$ from $1$ to $n$ do
(4)        if $(A[j] = 1)$ then return($REG[j]$) end if;
(5)    end for
end operation.
```

Fig. 16.13 From mem-to-mem-swap to consensus (code for process p_i)

A base SWMR register $REG[i]$ is associated with each process p_i. This register is used to make public the value proposed by p_i (line 1). A process p_j reads it at line 4 if it has to decide the value proposed by p_i.

There are $n + 1$ mem-to-mem-swap objects. The array $A[1..n]$ is initialized to $[0, \ldots, 0]$, while the object R is initialized to 1. The object $A[i]$ is written only by p_i, and this write is due to a mem-to-mem-swap operation: p_i exchanges the content of $A[i]$ with the content of R (line 2). As we can see, differently from $A[i]$, the mem-to-mem-swap object R can be written by any process. As described in lines 2–4, these objects are used to determine the decided value. After it has exchanged $A[i]$ and R, a process looks for the first entry j of the array A such that $A[j] \neq 0$, and decides the value deposited by the corresponding process p_j.

Remark: a simple invariant To better understand the algorithm, let us observe that the following relation remains invariant: $R + \Sigma_{1 \leq i \leq n} A[i] = 1$. As initially $R = 1$ and $A[i] = 0$ for each i, this relation is initially satisfied. Then, due to the fact that the binary operation $A[i]$.mem_to_mem_swap(R) issued at line 2 is atomic, it follows that the relation remains true forever.

Lemma 40 *The mem-to-mem-swap object type has consensus number n in a system of n processes.*

Proof The algorithm is trivially wait-free (the loop is bounded). As before, let the winner be the process p_i that sets $A[i]$ to 1 when it executes line 2. As any mem-to-mem-swap register $A[j]$ is written at most once, we conclude from the previous invariant that there is a single winner. Moreover, due to the atomicity of the mem-to-mem-swap objects, the winner is the first process that executes line 2. As, before becoming the winner, a process p_i has deposited into $REG[i]$ the value v it proposes to the consensus object, we have $REG[i] = v$ and $A[i] = 1$ before the other processes terminate the execution of line 2. It follows that all the processes that decide return the value proposed by the single winner process. $\square$

Remark Differently from the construction based on compare&swap objects, the construction based on mem-to-mem-swap objects uses the indexes and the number of processes.

Definition 5 *An object type is* universal in a system of n processes *if, together with atomic registers, it permits a wait-free implementation of a consensus object for n processes to be built.*

This notion of universality completes the one introduced in Chap. 14. It states that, in an n-process system, universal constructions can be built as soon as the base read/write system is enriched with an object type that is universal in a system of $m \geq n$ processes.

The following theorem is an immediate consequence of the previous lemma.

Theorem 76 *For any n, mem-to-mem-swap objects are universal in a system of n processes.*

```
operation $C$.propose($v$) is
        $Q$.enqueue($v$);
        $aux \leftarrow Q$.peek();
        return($aux$)
end operation.
```

Fig. 16.14 From an augmented queue to consensus

16.5.3 Consensus from an Augmented Queue

Augmented queue This object type is defined from a simple modification of a classical queue. An augmented queue is an atomic queue with an additional operation denoted peek() that returns the first item of the queue without removing it. Hence this type of queue allows a process to read a part of a queue without modifying it.

Interestingly, an augmented queue has infinite consensus number. This shows that simple modification of an object can increase its wait-free computability power to infinity.

From an augmented queue to consensus The algorithm in Fig. 16.14 is a wait-free construction of a consensus object from an augmented queue. The construction is fairly simple. The augmented queue Q is initially empty. A process first enqueues the value v it proposes to the consensus object. Then, it invokes Q.peek() to obtain the first value that has been enqueued. It is easy to see that, as the construction is based on a compare&swap object, this construction works for any number of processes and is independent of the number of processes.

The proof of the following theorem is trivial.

Theorem 77 *An augmented queue has an infinite consensus number.*

16.5.4 From a Sticky Bit to Binary Consensus

Sticky bit A sticky bit *SB* is an atomic register which is initialized to $\perp$ and can then contain forever either the value 0 or the value 1. It can be accessed only by the operation *SB*.write(v), where the value of the parameter v is 0 or 1.

The effect of the invocation of *SB*.write(v) can be described by the following sequence of statements executed atomically (X represents the current value of *SB*). The first invocation (as defined by the linearization order) gives its value to the sticky bit *SB* and returns the value *true*. Any other invocation returns *true* if the value v it wants to write is the value of X or *false* if $v \neq X$.

```
operation $SB$.write($v$) is
    if ($X = \perp$)
       then $X \leftarrow v$; return($true$)
       else  if ($X = v$) then return($true$) else return($false$) end if
    end if
end operation.
```

```
operation $C$.propose($v$) is
        $aux \leftarrow SB$.write($v$);
        if ($aux$) then $res \leftarrow v$ else $res \leftarrow (1 - v)$ end if;
        return($res$)
end operation.
```

Fig. 16.15 From a sticky bit to binary consensus

From a sticky bit to consensus A very simple construction of a binary consensus object from a sticky bit is described in Fig. 16.15. It is assumed that the values proposed are 0 and 1.

As we will see in the next chapter, a multi-valued consensus object can easily be built from binary consensus objects. The following theorem (whose proof is left to the reader) follows from this observation and the previous construction of a binary consensus object.

Theorem 78 *A sticky bit has an infinite consensus number.*

Remark Similarly to the wait-free constructions of a consensus object that are based on compare&swap or an augmented queue, the previous construction (based on a sticky bit) is independent of the number of processes, and consequently works for any value of n.

16.5.5 Impossibility Result

Corollary 9 *There is no wait-free implementation of an object of type compare& swap, mem-to-mem-swap, augmented queue, or sticky bit from atomic registers and base objects of type either stack, queue, priority queue, swap, fetch&add, or test&set.*

Proof The proof follows directly from the combination of Theorem 74 (the cited base objects have consensus number 2), Theorems 75–78 (compare&swap objects, mem-to-mem-swap objects, augmented queues, and sticky bits have infinite consensus number), and Theorem 68 (impossibility of wait-free implementing Y from X when $CN(X) < CN(Y)$). □

16.6 Hierarchy of Atomic Objects

16.6.1 From Consensus Numbers to a Hierarchy

Consensus numbers establish a hierarchy on the computability power of object types to wait-free implement consensus objects. Due to the very existence of universal constructions (Chap. 14), this hierarchy is actually on their power to wait-free implement any object defined by a sequential specification on total operations.

Table 16.1 Consensus hierarchy

Consensus number	Concurrent atomic objects
1	read/write registers, snapshot objects, M-renaming ($M \geq 2n-1$)
2	test&set, swap, fetch&add, FIFO queue, stack, list, set, $\cdots$
...	...
$2m-2$	m-register assignment ($m > 1$)
...	...
$+\infty$	compare&swap, LL/SC (linked load, store conditional), Augmented queue, sticky bit, mem-to-mem swap, $\cdots$

- This means that, while read/write registers are universal in failure-free systems, this is no longer true in asynchronous systems prone to any number of process crashes when one is interested in wait-free implementations. More generally, the synchronization primitives (provided by shared memory distributed machines) have different power in the presence of process crashes: compare&swap is stronger than test&set that is in turn stronger than atomic read/write operations.
- Interestingly, consensus numbers show also that, in a system of two processes, the concurrent versions of classical objects encountered in sequential computing such as stacks, lists, sets, and queues are as powerful as the test&set or fetch&add synchronization primitives when one is interested in providing processes with wait-free objects.

More generally, consensus numbers define an infinite hierarchy on concurrent objects. Let X be an object at level x, Y an object at level y, and $x < y$. This hierarchy is such that:

1. It is not possible to wait-free implement Y from objects X and atomic registers.
2. It is possible to wait-free implement X from objects Y and atomic registers.

This hierarchy is called a consensus hierarchy, wait-freedom hierarchy, or Herlihy's hierarchy. Some of its levels and corresponding objects are described in Table 16.1. (LL/SC registers have been defined in Sect. 6.3.2, m-register assignment objects are defined in the exercise section.)

16.6.2 On Fault Masking

The previous hierarchy shows that fault-masking can be impossible to achieve when the designer is provided with base atomic objects whose computability power (as measured by their consensus number) is too weak.

As an example, a wait-free FIFO queue that has to tolerate the crash of a single process cannot be built from atomic registers. This follows from the fact that the consensus number of a queue is 2 while the consensus number of atomic registers is 1.

More generally, considering the consensus hierarchy and a universal construction (as defined in Chap. 14), it follows that, in a system of n processes, wait-free implementations of atomic objects defined by a sequential specification on total operations require base objects whose consensus number is $m \geq n$.

16.6.3 Robustness of the Hierarchy

The previous hierarchy of concurrent objects is *robust* in the sense that any number of objects of the same deterministic type T_x whose consensus number is x cannot wait-free implement (with the additional help of any number of read/write registers) an object Y whose consensus number y is such that $y > x$.

16.7 Summary

This chapter has introduced the notions of consensus number and consensus hierarchy. These notions are related to the wait-free implementation of concurrent objects.

Important results related to the consensus hierarchy are: (a) atomic read/write registers, snapshot objects, and $(2n-1)$-renaming objects have consensus number 1, (b) test&set, fetch&add, queues, and stacks have consensus number 2 and (c) compare&swap, LL/SC, and sticky bits have consensus number $+\infty$.

Combined with a universal construction, the consensus hierarchy states which synchronization computability power is needed when one wants to be able to wait-free implement, in a system of n processes, *any* concurrent object defined by a sequential specification.

16.8 Bibliographic Notes

- The consensus number notion, the consensus hierarchy and wait-free synchronization are due to M. Herlihy [138].
- The impossibility of solving consensus from atomic registers in a system of two processes was proved in several papers [76, 199, 257].

  This impossibility result is the shared memory counterpart of the well-known result named FLP (from the names of the persons who proved it, namely M.J. Fischer, N.A. Lynch, and M.S. Paterson). This result is on the impossibility to solve consensus in asynchronous message-passing systems prone to (even a single) process crashes [102].
- The reader interested in synchronization primitives that exist (or could be realized) on multiprocessor machines can consult [117, 118, 157, 180, 260].

- The notion of a sticky bit is due to S. Plotkin [228].
- Developments on the multiplicative power (with respect to the number of process crash failures) of consensus numbers can be found in [159].
- The *robustness* of the consensus hierarchy addresses the following question: given any number of concurrent objects whose consensus numbers are x or lower than x, is it possible to build a concurrent object whose consensus number is greater than x? The hierarchy is robust if and only if the answer to this question is "no". It is shown in [55, 226] that the hierarchy containing only deterministic object types is robust. The reader interested in this issue can consult [166, 198, 256] for more developments.
- Numerous wait-free constructions of concurrent objects in different failure models (crash, omission, and Byzantine failures) are described in [169].

16.9 Exercises and Problems

1. Prove the construction of a consensus object which is based on a sticky bit (Fig. 16.15).
2. Prove that the consensus number of the LL/SC pair of primitives (defined in Sect. 6.3.2 is $+\infty$. (To that end an appropriate construction of a consensus object has to be designed and proved correct.)
3. Let R_1 and R_2 be two atomic registers. The mem-to-mem assignment primitive R_1.assign(R_2) copies atomically the content of R_2 into R_1 (the content of R_2 is not modified). Prove that its consensus number is $+\infty$. (Hint: use arrays whose size is the number of processes n.)

   Solution in [138].
4. The m-register assignment primitive is an assignment that is on m atomic registers at the same time. Let $v_1, \ldots, v_m$ be m values. $(R_1, \ldots, R_m)$.assign $(v_1, \ldots, v_m)$ atomically assigns v_i to R_i for each $i \in \{1, m\}$. Show that the consensus number of m-register assignment is $2m - 2$.

   Solution in [138].
5. Let us consider the following boosting of the operation X.test&set() (whose consensus number is 2). The extended operation X.ext_test&set() takes as input parameter a pointer P to an atomic read/write object. The invocation X.ext_test&set(P) has the same effect as the atomic execution of the following sequence of assignments
$$(P \downarrow) \leftarrow X;\ X \leftarrow 0.$$

Basically, X.ext_test&set(P) extends the X.test&set() atomic operation by encapsulating into it the writing of the register pointed to by P.

Show that the consensus number of ext_test&set() is $+\infty$.

Solution in [157].

6. Let us consider an asynchronous system of n processes prone to any number of process crashes where the processes cooperate with atomic read/write registers and consensus objects whose consensus number is equal to $(n-1)$.

   Show that it is impossible to design a consensus object that ensures wait-freedom for one process and obstruction-freedom for the $(n-1)$ other processes.

   Solution in [164].

7. Build a wait-free safe register from a single test&set register.

   Solution in [169].

Chapter 17
The Alpha(s) and Omega of Consensus: Failure Detector-Based Consensus

Chapter 16 has shown that a base read/write system enriched with objects whose consensus number is at least x allows consensus objects to be wait-free implemented in a system of at most x processes. Moreover, thanks to universal constructions (Chap. 14) these objects allow construction of wait-free implementations of any object defined by a sequential specification on total operations.

Hence, the question: is the enrichment of a read/write asynchronous system with stronger operations such as compare&swap the only means to obtain wait-free implementations of consensus objects? This chapter shows that the answer to this question is "no". To that end it presents another approach which is based on information on failures (the *failure detector*-based approach). Each process is enriched with a specific module that gives it (possibly unreliable) information on the faulty/correct status of processes.

The chapter presents first a de-construction of compare&swap that allows us to introduce two types of objects whose combination allows consensus to be solved despite asynchrony and any number of process crashes. One is related to the safety of consensus, the other one to its liveness. The chapter presents then the failure detector Ω which has been proved to be the one providing the weakest information on failure which allows consensus to be solved. Ω is used to ensure the consensus termination property. Several base objects that can be used to guarantee the consensus safety properties are then introduced. Differently from Ω, these objects can be wait-free built from read/write registers only. The chapter finally investigates additional synchrony assumptions that allow Ω to be implemented.

Keywords Active disk · Adopt-commit · Alpha abstraction · Atomic register · Compare&swap · Consensus object · Indulgence · Omega abstraction · Regular register · Round-based abstraction · Shared disk · Store-collect · Timing assumption

17.1 De-constructing Compare&Swap

As noticed in Chap. 14, a consensus object *CONS* can be seen as a write-once register whose value is determined by the first invocation of its operation *CONS*.propose(). Moreover, any invocation of *CONS*.propose() (which terminates) returns the value of the consensus object.

From a trivial failure-free case Let us consider a failure-free asynchronous system made up of processes cooperating through atomic registers. A trivial consensus algorithm consists in deciding the value proposed by a predetermined process, say p_ℓ. That process (which can be any predetermined process) deposits the value it proposes into an SWMR register *DEC* (initialized to $\perp$, a default value that cannot be proposed), and a process $p_i \neq p_\ell$ reads the shared register *DEC* until it obtains a value different from $\perp$. Assuming that p_ℓ eventually proposes a value, this constitutes a trivial starvation-free implementation of a consensus object. (Let us notice that the shared register is not required to be atomic. It could be a regular register.)

First to compare&swap An attempt to adapt this algorithm to a system where any number of processes may crash and in which not all the processes propose a value could be the following. Let the "leader" be the process that "imposes" a value as the decided value. As there is no statically defined leader p_ℓ that could impose its value on all (any static choice could select a process that crashes before writing into *DEC*), any process is entitled to compete to play the leader role. Moreover, to ensure the wait-freedom property of the algorithm implementing the operation *CONS*.propose(), as soon as a process proposes a value, it must try to play the leader role just in case no other process writes into *DEC* (because it crashes before or never invokes *CONS*.propose()).

So, when it proposes a value, a process p_i first reads *DEC*, and then writes the value v it proposes if its previous read returned $\perp$. Unfortunately, between a read obtaining the value $\perp$ and the subsequent write into *DEC* by a process p_i, the value of *DEC* may have been changed from $\perp$ to some value v by another process p_j (and maybe v has then been read and decided by other processes). Hence, this naive approach does not work (we already knew that because the consensus number of atomic registers is 1).

A way to solve the previous problem consists in forging an atomic operation that includes both the read and the associated conditional write. As we have seen in Chap. 16, this is exactly what is done by the operation compare&swap(). If the register *DEC* is provided with that operation, *CONS*.propose(v) is implemented by *DEC*.compare&swap($\perp$, v) and the first process that invokes it becomes the dynamically defined leader that imposes the value v as the value of the consensus object. (The term "winner" was used in Chap. 16 instead of the term "leader"). It is important to notice that, as it is atomic, the operation compare&swap() allows one to cope both with process crashes and the fact that not all processes propose a value. This intuitively explains why the consensus number of a compare&swap register is $+\infty$.

Conceptually, the first process that successfully invokes *DEC*.compare&swap (⊥, v) is atomically elected as the leader which imposes a value as the value decided by the consensus object. This notion of "leader" appears only at the operational level: the leader is the process whose invocation of *DEC*.compare&swap() appears first in the linearization order associated with the atomic object *DEC*. (Let us observe that, from the consensus object construction point of view, the fact that the leader crashes after having deposited a value in the consensus object is unimportant).

And then de-constructing compare&swap (Dissociating the leader election from the deposit of a value) The previous discussion suggests that, when we have to implement a consensus object without the help of a compare&swap register, we could try to first decouple the two functions we have just identified (namely the election of a leader and the deposit of a value) and then combine them to build a wait-free consensus algorithm. The underlying idea is that a process tries to deposit the value it proposes into the consensus object only if it considers it is a leader. This chapter shows that this idea works. It presents two types of abstractions, called *alpha* and *omega*, that allow construction of a wait-free consensus object. An alpha object is related to the deposit of a value proposed by a process while guaranteeing the consensus safety properties, while an omega object provides the processes with a leader service that allows any invocation of *CONS*.propose() issued by a correct process to terminate (Fig. 17.1).

As, on one side, the determination of a leader and the deposit of a value are now two separate operations and, on another side, the leader can be needed to help decide a value, an additional assumption is required, namely the leader is required to participate in the algorithm. (As we can notice, this assumption is de facto always satisfied when using a compare&swap object to wait-free construct a consensus object.)

From alpha + omega to consensus After having defined three (safety-related) alpha abstractions and the (liveness-related) omega abstraction, this chapter presents wait-free implementations of consensus objects based on these abstractions. It then shows how each alpha abstraction can be built from shared read/write registers only while the construction of the omega abstraction requires that the underlying system satisfies additional synchrony assumptions.

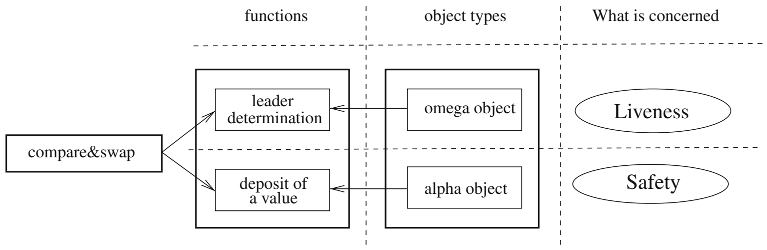


Fig. 17.1 From compare&swap to alpha and omega

17.2 A Liveness-Oriented Abstraction: The Failure Detector Ω

The notion of a failure detector, which is due to D.P. Chandra and S. Toueg (1996), was introduced in Sect. 5.3.1, (where the eventually restricted leader failure detector Ω_X and the eventually perfect failure detector $\Diamond P$ were introduced).

Let us remember that a failure detector is a device (object) that provides each process with a read-only local variable which contains (possibly incomplete and unreliable) information related to failures. A given class of failure detector is defined by the type and the quality of this information.

A failure detector *FD* is *non-trivial* with respect to a system *S* and an object *O* if the object *O* can be wait-free implemented in *S* enriched with *FD* while it cannot in *S* alone.

17.2.1 Definition of Ω

The failure detector Ω was introduced by T.D. Chandra, V. Hadzilacos, and S. Toueg (1996), who have showed that this failure detector captures the weakest information on failures that allows one to build a consensus object in an asynchronous system prone to process crash failures. As there is no wait-free implementation of a consensus object in asynchronous systems where (a) processes communicate by read/write registers only and (b) any number of processes may crash (as shown in Chap. 16), it follows that Ω is a non-trivial failure detector. Consequently (as we will see in Sect. 17.7), Ω cannot be implemented in a pure asynchronous system.

Ω is sometimes called an *eventual leader oracle*. From a more practical point of view, it is sometimes called an *eventual leader* service.

Definition of Ω This failure detector provides each process p_i with a local variable denoted $leader_i$ that p_i can only read. These local variables satisfy the following properties (let us remember that a process is correct in an execution if it does not crash in that execution):

- Validity. For any p_i and at any time, $leader_i$ contains a process index (i.e., $leader_i \in \{1, \ldots, n\}$).
- Eventual leadership. There is a finite time after which the local variables $leader_i$ of the correct processes contain forever the same index ℓ and the corresponding process p_ℓ is a correct process.

Taking $\Pi = \{1, \ldots, n\}$ (the whole set of process indexes) Ω corresponds to Ω_Π, the failure detector introduced in Sect. 5.3.1. As we have seen in that chapter, this means that there is a finite anarchy period during which processes can have arbitrary leaders (distinct processes having different leaders and some leaders being crashed processes). But this anarchy period eventually terminates, and after it has terminated, all correct processes have "forever" the same correct leader. However, it is important to notice that no process knows when the anarchy period terminates and the eventual leader is elected.

The term "forever" is due to asynchrony. As there is no notion of physical time accessible to the processes, once elected, the leader process has to remain leader for a long enough period in order to be useful. Even if this period is finite, it cannot be bounded. Hence the term "forever".

A formal definition Let us consider a discrete global clock, not accessible to the processes, whose domain is the set of integers, denoted N. As previously indicated, let Π be the set of the process indexes.

A *failure pattern* is a function $F : N \rightarrow 2^{\Pi}$ such that $F(\tau)$ denotes the set of processes that have crashed by time τ. As a crashed process does not recover, we have $(\tau \leq \tau') \Rightarrow \big(F(\tau) \subseteq F(\tau')\big)$. The set C of processes that are correct in a given execution is the set of processes that do not crash during that execution, i.e., $C = \Pi \setminus \big(\cup_{\tau \geq 1} F(\tau)\big)$.

$\forall i \in \Pi, \forall \tau \in N$, Let $leader_i^{\tau}$ be the value of $leader_i^{\tau}$ at time τ. Let C_F be the set of indexes of the correct processes in the failure pattern $F()$. With these notations, Ω is defined as follows:

- Validity. $\forall F(), \forall i \in \Pi, \forall \tau \in N : \ leader_i^{\tau} \in \Pi$.
- Eventual leadership. $\forall F(), \exists \ell \in C_F, \exists \tau \in N : \forall \tau' \geq \tau : \forall i \in C_F : \ leader_i^{\tau'} = \ell$.

Ω is a failure detector The previous definition makes clear the fact that Ω is a failure detector. This comes from the fact that it is defined from a failure pattern. When it reads $leader_i = x$ at some time τ, a process p_i learns from the failure detector Ω that p_x is its current leader. As just seen, this information may be unreliable only during a finite period of time. At any time, p_i knows that (a) $leader_i$ will contain forever the index of a correct process and (b) all these local variables will contain the same process index.

17.2.2 Ω-Based Consensus: Ω as a Resource Allocator or a Scheduler

As indicated in Sect. 17.1, an Ω-based implementation of a consensus object *CONS* relies on Ω to ensure the wait-freedom property of the consensus object (i.e., the invocation *CONS*.propose() by a correct process always terminates).

An alpha abstraction (see below) will be used to ensure the consensus safety properties (validity and agreement). Using only alpha, it would be possible for an invocation of *CONS*.propose() to never terminate. If no value is decided before the (unknown but finite) time when a common correct leader is elected forever, Ω can be used in order for a single process to continue executing "useful" steps and impose a decided value on all the processes. In that sense, the failure detector Ω can be seen as a resource allocator (or a scheduler).

17.3 Three Safety-Oriented Abstractions: Alpha$_1$, Alpha$_2$, and Alpha$_3$

An alpha abstraction is an object that, in combination with Ω, allows one to build a wait-free implementation of a consensus object; As announced previously, while Ω is used to ensure the liveness of the consensus object, an alpha object is used to ensure its safety property. These implementations of a consensus object are based on distributed iterations where an iteration step is called a *round*.

This section presents three alpha abstractions; two are round-free, while the other one is round-based. *Round-free* means that the definition of the corresponding alpha object does not involve the round notion while *round-based* means that the corresponding alpha object involves the notion of round.

17.3.1 A Round-Free Abstraction: Alpha$_1$

Such an alpha object, called here alpha$_1$, was introduced by E. Gafni (1998) (under the name *adopt-commit* object).

Alpha$_1$ is a one-shot object that provides the processes with a single operation denoted adopt_commit(). This operation takes a value as input parameter (we then say that the invoking process *proposes* that value) and returns a pair $\langle d, v\rangle$, where d is a control tag and v a value (we then say that the invoking process decides the pair $\langle d, v\rangle$).

Definition Let $ALPHA1$ be an alpha$_1$ object. Its behavior is defined by the following properties:

- Termination. An invocation of $ALPHA1$.adopt_commit() by a correct process terminates.
- Validity. This property is made up of two parts:
  - Output domain. If a process decides $\langle d, v\rangle$ then $d \in \{commit, adopt, abort\}$ and v is a value that was proposed to $ALPHA1$.
  - Obligation. If all the processes that invoke $ALPHA1$.adopt_commit() propose the same value v, then the only pair that can be decided is $\langle commit, v\rangle$.
- Quasi-agreement. If a process decides $\langle commit, v\rangle$ then any other deciding process decides $\langle d, v\rangle$ with $d \in \{commit, adopt\}$.

Intuitively, an alpha$_1$ object is an abortable consensus object. A single value v can be committed. The control tags *adopt* or *abort* are used to indicate that no value can be committed. It is easy to see (quasi-agreement property) that, if a process decides $\langle abort, -\rangle$, no process decides $\langle commit, -\rangle$. Differently, if a process decides $\langle adopt, -\rangle$, it cannot conclude which control tag was decided by other processes.

17.3.2 A Round-Based Abstraction: Alpha$_2$

This object, called here alpha$_2$, was introduced by L. Lamport (1998). It is a round-based object which takes a round number as a parameter. Hence, differently from the previous alpha$_1$ object, an alpha$_2$ object is a multi-shot object and a single instance is required to implement a consensus object.

An alpha$_2$ object provides the processes with a single operation denoted deposit() which takes two input parameters, a round number r and a proposed value v (v is the value that the invoking process wants to deposit into the alpha$_2$ object). An invocation of deposit() returns a value which can be the default value $\perp$ (that cannot be proposed by a process). Moreover, it is assumed that (a) no two processes use the same round numbers and (b) a process uses a given round number only once and its successive round numbers are increasing.

Definition Let $ALPHA2$ be an alpha$_2$ object. Its behavior is defined by the following properties:

- Termination. An invocation of $ALPHA2$.deposit() by a correct process terminates.
- Validity. This property is made up of two parts:
  - Output domain. An invocation $ALPHA2$.deposit$(r, -)$ returns either $\perp$ or a value v such that $ALPHA2$.deposit$(-, v)$ has been invoked by a process.
  - Obligation. If there is an invocation $I = ALPHA2$.deposit$(r, -)$ such that any invocation $I' = ALPHA2$.deposit$(r', -)$ that started before I terminates is such that $r' < r$, then I returns a non-$\perp$ value.
- Quasi-agreement. Let $ALPHA2$.deposit(r, v) and $ALPHA2$.deposit(r', v') be two invocations that return w and w', respectively. We have $\big((w \neq \perp) \wedge (w' \neq \perp)\big) \Rightarrow (w = w')$.

It is easy to see (quasi-agreement property) that an alpha$_2$ object is a weakened consensus object in the sense that two processes cannot decide different non-$\perp$ values.

An example To illustrate the round numbers and the obligation property, let us consider Fig. 17.2, where there are six processes, among which the process p_4 has (initially) crashed. The invocations of $ALPHA2$.deposit() are indicated by double-

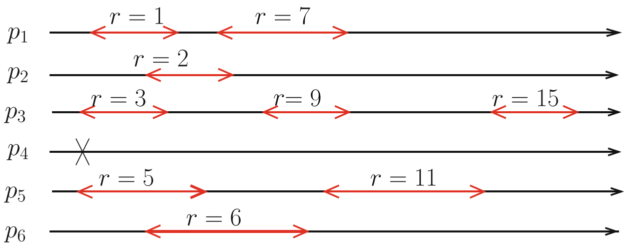


Fig. 17.2 Obligation property of an alpha$_2$ object

arrow segments, the round number associated with an invocation being indicated above the corresponding segment.

As required by alpha$_2$, no two invocations of $ALPHA2$.deposit() use the same round number and the consecutive round numbers used by each process are increasing. When we consider all the invocations with round number smaller than 11, it is possible that they all return $\perp$. Indeed, these invocations satisfy the termination, obligation, and quasi-agreement properties stated above. As any invocation with a round number smaller than 11 is concurrent with another invocation with a greater round number, the obligation property allows them to return the default value $\perp$.

The situation is different for the invocation $I = ALPHA2$.deposit$(11, -)$ issued by p_5. As there is no invocation with a greater round number that started before I terminates, the obligation property states that I must return a non-$\perp$ value, namely a value v such that $deposit(r, v)$ was invoked with $r \leq 11$.

Due to the very same obligation property, the invocation $I = ALPHA2$.deposit $(15, -)$ issued by p_3 also has to return a non-$\perp$ value. Moreover, due to the quasi-agreement property, that value has to be the same as the one returned by I.

17.3.3 Another Round-Free Abstraction: Alpha$_3$

Another abstraction that can be used to maintain the safety of a consensus object is a *store-collect* object. Such an object was introduced in Sect. 7.2. and the notion of a *fast store-collect* object was introduced in Sect. 7.3.

Reminder: store-collect object Let $ALPHA3$ be a store-collect object. As we have seen, such an object provides the processes with two operations. The first one, denoted store(), allows the invoking process p_i to deposit a new value into $ALPHA3$ while discarding the previous value it has previously deposited (if any). The second operation, denoted collect(), returns to the invoking process a set of pairs $\langle j, val\rangle$, where val is the last value deposited by the process p_j. The set of pairs returned by an invocation of collect() is called a $view$.

As we have seen in Chap. 7, a store-collect object has no sequential specification. We have also seen that such an object has an efficient adaptive wait-free implementation based on read/write registers. More precisely, the step complexity (number of shared memory accesses) of the operation collect() is $O(k)$, where k is the number of different processes which have invoked the operation store() (Theorem 30, Chap. 7). The step complexity of the first invocation of the operation store() by a process p_i is also $O(k)$, while the step complexity of its next invocations of store() is $O(1)$ (Theorem 29, Chap. 7).

Reminder: fast store-collect object Such an object is a store-collect object in which the operations store() and collect() are merged to form a single operation denoted store_collect(). Its effect is that of an invocation of store() immediately followed by an invocation of collect(). This operation is not atomic.

We have seen that the step complexity of such an operation is $O(1)$ when, after some time, a single process invokes that operation (Theorem 32, Chap. 7).

17.3.4 The Rounds Seen as a Resource

Let us consider that, according to the alpha object that is used, due to Ω there is a time τ after which only one process invokes repeatedly $ALPHA1$.adopt_commit() or $ALPHA2$.deposit() or $ALPHA3$.store_collect().

As we will see in the next section, the role of Ω is to be a round allocator in such a way that after some finite time, a single process executes rounds. According to the object that is used, this will allow an invocation of $ALPHA1$.adopt_commit() to return $\langle commit, v\rangle$, or an invocation of $ALPHA2$.deposit() to return a value $v \neq \bot$, or an invocation of $ALPHA3$.store_collect() to return a set which allows the invoking process to decide.

17.4 Ω-Based Consensus

Let us remember that each process is endowed with a read-only local variable $leader_i$ whose content is supplied by the failure detector Ω. Let $CONS$ be the consensus object we want to build.

The three consensus algorithms described below assume that the common correct leader eventually elected by Ω invokes $CONS$.propose(v). The case where this assumption is not required is considered in Sect. 17.4.4.

17.4.1 Consensus from Alpha$_1$ Objects and Ω

Internal representation of the consensus object The internal representation of $CONS$ consists of an atomic register and an unbounded array of alpha$_1$ objects:

- DEC is an MWMR atomic register initialized to the default value $\bot$. Its aim is to contain the value decided by the consensus object.
- $ALPHA1[1..]$ is an unbounded array of alpha$_1$ objects. $ALPHA1[r]$ is the alpha$_1$ object used by a process when it executes its rth round. As we have seen, an invocation $ALPHA1[r]$..commit_adopt() returns a pair res such that $res.tag \in \{commit, adopt, abort\}$ and $res.val$ contains a proposed value.

Each process p_i manages two local variables: est_i, which contains its current estimate of the decision value, and r_i, which is its current round number.

```
operation $CONS$.propose($v_i$) is
(1)  $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
(2)  while $(DEC = \bot)$ do
(3)     if $(leader_i = i)$ then
(4)        $r_i \leftarrow r_i + 1$;
(5)        $res_i \leftarrow ALPHA1[r]$.commit_adopt($est_i$);
(6)        if $res_i = \langle commit, - \rangle$
(7)           then $DEC \leftarrow res_i.val$
(8)           else $est_i \leftarrow res_i.val$
(9)        end if
(10)    end if
(11) end while;
(12) return($DEC$)
end operation.
```

Fig. 17.3 From alpha$_1$ (adopt-commit) objects and Ω to consensus

The algorithm implementing the operation *CONS*.propose() This simple algorithm is described in Fig. 17.3. The processes execute asynchronous rounds until they decide. It is important to notice that only the processes participating in the consensus execute rounds and the participating processes are not necessarily executing the same round at the same time.

A process p_i first deposits the value v_i it proposes into its estimate of the decision value est_i (line 1). Then, it loops until it reads $DEC \neq \bot$ (line 2). When this occurs it returns the value of DEC (line 12).

If $DEC = \bot$, a process p_i first check if it is a leader (line 2). If it is not, it re-enters the **while** loop. Otherwise, it tries to impose its current value of est_i as the decision value. To that end it proceeds to its next round r (line 4) and invokes $ALPHA1[r]$.commit_adopt(est_i) (line 5) to benefit from the help of the alpha$_1$ object associated with this round. If it obtains a pair $\langle commit, v\rangle$ from its invocation (line 6), p_i writes v into DEC (line 7) and decides. If it does not obtain the tag *commit*, p_i adopts the value v it has just obtained from $ALPHA1[r]$ as its new estimate value est_i before re-entering the **while** loop.

Theorem 79 *Let us assume that the eventual leader elected by Ω participates (i.e., invokes CONS*.propose()*). The construction described in Fig.* 17.3 *is a wait-free implementation of a consensus object.*

Proof The consensus validity property states that a decided value is a proposed value. The proof of this property follows from the following observations. Initially, the estimates est_i of the participating processes contain proposed values (line 1). Then, the input parameter of any invocation of commit_adopt() is such an estimate, and thanks to the "output domain" validity property of the alpha$_1$ objects, the value returned $res_i.val$ is also an estimate of a proposed value (line 5). Hence, only proposed values can be assigned to DEC (line 7), and consequently, a decided value is a value proposed by a participating process.

The consensus agreement property states that no two processes can decide different values. Let us consider the first round r during which a process assigns a value to the atomic register DEC (line 7). Let p_i be a process that issues such an assignment. It follows from lines 5–6 that p_i has obtained $res_i = \langle commit, v\rangle$ from $ALPHA1[r]$. Moreover, it follows from the quasi-agreement property of this alpha$_1$ object that any other process p_j that returns from $ALPHA1[r]$.adopt_commit() obtains $res_j = \langle commit, v\rangle$ or $res_j = \langle adopt, v\rangle$. If p_i obtains $\langle commit, v\rangle$, it writes v into the atomic register DEC, and if it obtains $\langle adopt, v\rangle$, it writes v into est_i. It follows that, any process that executes round r either decides before entering round $r+1$ or starts round $r+1$ with $est_j = v$. This means that, from round $r+1$, the only value that a process can propose to $ALPHA1[r+1]$ is v. It follows that, from then on, only v can be written into the atomic register DEC, which concludes the proof of the consensus agreement property.

The consensus termination property states that any invocation of $CONS$.propose() by a correct process terminates. Due to the termination property of the alpha$_1$ objects, it follows that, whatever r, any invocation of $ALPHA1[r]$.adopt_commit() terminates. It follows that the proof of the consensus termination property amounts to showing that a value is eventually written in DEC. We prove this by contradiction.

Let us assume that no value is ever written in DEC. It follows from the eventual leadership property of Ω that there is a finite time τ_1 after which there is a correct process p_ℓ that is elected as permanent leader. Moreover, there is a time τ_2 after which only correct processes execute the algorithm. It follows that there is a time $\tau \geq \max(\tau_1, \tau_2)$ when all the processes p_i that have invoked $CONS$.propose() and have not crashed are such that $leader_i = \ell$ remains true forever. It follows that, after τ, p_ℓ is the only process that repeatedly executes lines 4–10. As (by assumption) it never writes into DEC, p_ℓ executes an infinite number of rounds. Let r be a round number strictly greater than any round number attained by any other process. It follows that p_ℓ is the only process that invokes $ALPHA1[r]$.adopt_commit(est). It then follows from the obligation property of $ALPHA1[r]$ that this invocation returns the pair $\langle commit, est\rangle$. Hence, p_ℓ executes line 7 and writes est into DEC, which contradicts the initial assumption and concludes the proof of the termination property. $\square$

17.4.2 Consensus from an Alpha$_2$ Object and Ω

Internal representation of the consensus object The consensus object is represented in the read/write shared memory by an atomic register DEC (as in the previous section) and a single instance of an alpha$_2$ object denoted $ALPHA2$. Let us remember that such an object is a round-based object.

The algorithm implementing the operation $CONS$.propose() This algorithm, which is described in Fig. 17.4, is very simple. As previously, the processes execute asynchronous rounds.

```
operation $CONS$.propose($v_i$) is
(1)   $r_i \leftarrow i - n$;
(2)   while $(DEC = \bot)$ do
(3)       if $(leader_i = i)$ then
(4)           $r_i \leftarrow r_i + n$;
(5)           $res \leftarrow ALPHA2$.deposit$(r_i, v_i)$;
(6)           if $res \neq \bot$ then $DEC \leftarrow res$ end if
(7)       end if
(8)   end while;
(9)   return$(DEC)$
end operation.
```

Fig. 17.4 From an alpha$_2$ object and Ω to consensus

Let us remember that no two processes are allowed to use the same round numbers, and round numbers used by a process must increase. To that end, the process p_i is statically assigned the rounds $i, n+i, 2n+i$, etc. (where n is the number of processes). The code of the algorithm is very similar to the code based on alpha$_1$ objects. In this algorithm, *res* is not a pair; it contains the value returned by *ALPHA*2.deposit(), which is a proposed value or the default value $\bot$.

Theorem 80 *Let us assume that the eventual leader elected by Ω participates (i.e., invokes CONS*.propose()*). The construction described in Fig. 17.4 is a wait-free implementation of a consensus object.*

Proof The consensus validity and agreement properties follow directly from the properties of the alpha$_2$ object. More precisely, we have the following. Let us first observe that, due to the test of line 2, $\bot$ cannot be decided. The consensus validity property follows then from the "output domain" validity property of the alpha$_2$ object and the fact that a process p_i always invokes *ALPHA*2.deposit(r_i, v_i), where v_i is the value it proposes to the consensus object. The consensus agreement property is a direct consequence of the fact that $\bot$ cannot be decided combined with the quasi-agreement property of the alpha$_2$ object.

The proof of the consensus termination property is similar to the previous one. It follows from (a) the eventual leadership property of Ω, (b) the fact that the eventual leader proposes a value, and (c) the fact that, if no process has deposited a value into *DEC* before, there is a time τ after which the eventual leader is the only one to execute rounds. The proof is consequently the same as in Theorem 79. □

17.4.3 Consensus from an Alpha$_3$ Object and Ω

Internal representation of the consensus object The consensus object is represented in the read/write shared memory by an atomic register *DEC* (as in the previous sections) and a single instance of an alpha$_3$ (store-collect or fast store-collect) object denoted *ALPHA*3. Let us remember that such an object is a round-free object.

The algorithm implementing the operation *CONS*.propose() As in the two previous Ω-based algorithms, this algorithm is a round-based. It is described in Fig. 17.5. A process p_i invokes *CONS*.propose(v_i), where v_i is the value it proposes. Its invocation terminates when it executes the statement return(*DEC*), where *DEC* contains the value it decides (line 17).

The local variable r_i contains the current round number of p_i, while est_i contains its current estimate of the decision value (these local variables are initialized at line 1). A process executes a while loop (lines 2–16) until it decides (or crashes). Moreover, it executes the loop body (lines 4–14) only if it is currently considered as a leader by Ω (predicate of line 3).

When it is considered as a leader, p_i does the following. First it stores its current local state $\langle r_i, est_i\rangle$ into the store-collect object *ALPHA*3 and then reads its current content by invoking *ALPHA*3.collect() (line 4). (Let us observe that, if one wants to use a fast store-collect object instead of a more general store-collect object, line 4 is replaced by the statement $mem_i \leftarrow$ *ALPHA*3.store_collect($\langle r_i, est_i\rangle$).) Let us notice that line 4 is the only line where p_i accesses the store-collect object, i.e., the part of the shared memory related to the consensus safety property. All the other statements executed by p_i in a round (except the write into *DEC* if it decides) are local statements.

Then, p_i extracts into mem_i the pairs $\langle r, v\rangle$ contained in the view $view_i$ it has obtained (line 5) and computes the greatest round $rmax_i$ that, from its point of view, has ever been attained (line 6). Its behavior then depends then on whether it is or is not late with respect to $rmax_i$:

```
operation $CONS$.propose($v_i$) is
(1)  $r_i \leftarrow 1$; $est_i \leftarrow v_i$;
(2)  while ($DEC = \bot$) do
(3)    if ($leader_i = i$) then
(4)      $ALPHA3$.store($\langle r_i, est_i\rangle$); $view_i \leftarrow ALPHA3$.collect();
(5)      $mem_i \leftarrow \{ \langle r, v\rangle \mid (-, \langle r, v\rangle) \in view_i \}$;
(6)      $rmax_i \leftarrow \max\{r \mid \langle r, -\rangle \in mem_i\}$;
(7)      if ($r_i = rmax_i$)
(8)        then $set_i \leftarrow \{v \mid \langle r, v\rangle \in mem_i$ where $r \in \{rmax_i, rmax_i - 1\}\}$;
(9)             if ($r_i > 1) \wedge (set_i = \{est_i\})$
(10)              then $DEC \leftarrow est_i$
(11)              else $r_i \leftarrow r_i + 1$
(12)            end if
(13)       else $est_i \leftarrow v$ such that $\langle rmax_i, v\rangle \in mem_i$;  $r_i \leftarrow rmax_i$
(14)     end if
(15)   end if
(16) end while;
(17) return($DEC$)
end operation.
```

Fig. 17.5 From an alpha$_3$ (store-collect) object and Ω to consensus

- If it is late ($r_i < rmax_i$), p_i jumps to round $rmax_i$ and adopts as a new estimate a value that is associated with $rmax_i$ in the view it has previously obtained (line 13).
- If it is "on time" from a round number point of view ($r_i = rmax_i$), p_i checks if it can write a value into *DEC* and decide. To that end, it executes lines 8–12. It first computes the set set_i of the values that are registered in the store-collect object with round number $rmax_i$ or $rmax_i - 1$, i.e., the values registered by the processes that (from p_i's point of view) have attained one of the last two rounds.

  If it has passed the first round ($r_i > 1$) and its set set_i contains only the value kept in est_i, p_i writes it into *DEC* (line 10), just before going to decide at line 17. If it cannot decide, p_i proceeds to the next round without modifying its estimate est_i (line 11).

Hence, the base principle on which this algorithm rests is fairly simple to state. (It is worth noticing that this principle is encountered in other algorithms that solve other problems such as termination detection of distributed computations.) This principle can be stated as follows: processes execute asynchronous rounds (observation periods) until a process sees two consecutive rounds in which "nothing which is relevant has changed".

Particular cases It is easy to see that, when all processes propose the same value, no process decides in more than two rounds whatever the pattern of failure and the behavior of Ω. Similarly, only two rounds are needed when Ω elects a correct common leader from the very beginning. In that sense, the algorithm is optimal from the "round number" point of view.

On the management of round numbers In adopt-commit-based or alpha-based consensus algorithms, the processes that execute rounds execute a predetermined sequence of rounds. More precisely, in the adopt-commit-based consensus algorithm presented in Fig. 17.3, each process that executes rounds does execute the predetermined sequence of rounds numbered 1, 2, etc. Similarly, in the alpha$_1$-based consensus algorithm presented in Fig. 17.4, each process p_i that executes rounds executes the predetermined sequence of rounds numbered i, $i + n$, $i + 2n$, etc.

Differently, the proposed algorithm allows a process p_i that executes rounds to jump from its current round r_i to the round $rmax_i$, which can be arbitrarily large (line 13). These jumps make the algorithm particularly efficient. More specifically, let us consider a time τ of an execution such that (a) up to time τ, when a process executes line 9, the decision predicate is never satisfied, (b) processes have executed rounds and mr is the last round that was attained at time τ, (c) from time τ, Ω elects the same correct leader p_ℓ for any process p_i, and (d) p_ℓ starts participating at time τ. It follows from the algorithm that p_ℓ executes the first round, during which it updates r_ℓ to mr, and then, according to the values currently contained in *ALPHA*3, at most the rounds mr and $mr + 1$ or the rounds mr, $mr + 1$, and $mr + 2$. As the sequence of rounds is not predetermined, p_ℓ saves at least $mr - 2$ rounds.

17.4.4 When the Eventual Leader Elected by Ω Does Not Participate

When the process p_ℓ eventually elected as common leader by Ω does not participate, the termination property of the three previous consensus algorithms can no longer be ensured in all executions.

When the subset of processes that participate in the consensus algorithm can be any non-empty subset of processes, the failure detector Ω has to be replaced by the failure detector Ω_X introduced in Sect. 5.3.1.

Reminder: the failure detector Ω_X 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.

Modifying the consensus algorithms As an example we consider the consensus algorithm described in Fig. 17.5 (the two other Ω-based consensus algorithms can be modified in the same way). When the participating processes can be any subset of processes, the system is enriched with a failure detector Ω_X for any non-empty subset X of process indexes and the algorithm has to be modified as follows:

- A new array, denoted $PART[1..n]$, made up of Boolean SWMR atomic registers is introduced. This array is initialized to $[\mathit{false}, \ldots, \mathit{false}]$. Moreover, the statement $PART[i] \leftarrow \mathit{true}$ is added to line 1 to indicate that, from now on, p_i is a participating process.
- The statement $X \leftarrow \{j \mid PART[j]\}$ is introduced between line 3 and 4. Hence, X denotes the set of participating processes as currently known by p_i.
- Finally the predicate of line 3 is replaced by $\mathit{ev\_leader}_i(X) = i$; i.e., p_i checks if it is leader among the processes it sees as participating processes.

The extended algorithms are correct The fact that this extended algorithm is correct follows from the correction of the base algorithm plus the following two observations:

1. The fact that two processes p_i and p_j, while executing the same round r, are such that $\mathit{ev\_leader}_i(X) = i$ and $\mathit{ev\_leader}_j(X') = j$ with $X \neq X'$ does not create a problem. This is because the situation is exactly as if $X = X'$ and Ω_X has not yet stabilized to a single leader. Hence, the consensus safety property cannot be compromised.

2. The consensus termination property cannot be compromised for the following reason. There is a finite time after which each participating process p_i has set its Boolean $PART[i]$ to the value *true*. When this has occurred, all the correct participating processes have the same set X and, due to the restricted eventual leadership property of Ω_X, one of them will be elected as their common leader.

17.4.5 The Notion of an Indulgent Algorithm

Indulgent algorithm: definition Let us assume a wait-free algorithm A that implements an object R from base objects of type $T_1, \ldots, T_x$. The algorithm A is said to be *indulgent* with respect to a base type T, if assuming all the objects of the types different from T behave according to their specification, A never violates its safety properties when the objects of type T behave arbitrarily. This means that an indulgent algorithm can lose only its liveness property when some base objects do not behave according to their specification.

Ω-based algorithms are inherently indulgent Interestingly, the algorithms described in Figs. 17.3, 17.4 and 17.5 that provide wait-free constructions of a consensus object from alpha objects and an Ω (or Ω_X) object are indulgent with respect to Ω (or Ω_X). If the eventual leader property is never satisfied, it is possible that no invocation of $CONS$.propose() ever terminates, but if they terminate despite the arbitrary behavior of Ω, the consensus safety properties (validity and agreement) are satisfied.

Differently, these constructions are not indulgent with respect to the alpha object type (which is used to ensure the safety property of the constructed consensus object). Indulgence can only be with respect to object types whose definition includes "eventual" properties (i.e., properties such hat "there is a time after which …"). The objects with respect to which an algorithm can be indulgent are objects used to ensure termination properties. No indulgence is possible with respect to the objects used to ensure safety properties. The notion of indulgence can be extended to the assumption an algorithm relies on. The previous constructions are also indulgent with respect to the assumption that the eventual leader participates in the consensus. If that process never proposes a value, the only "bad thing" that can happen is that the invocations of $CONS$.propose() do not terminate.

17.4.6 Consensus Object Versus Ω

When the failure detector Ω is used to implement a consensus object, no process ever knows from which time instant the failure detector provides forever the processes with the index (identity) of the same correct process. A failure detector is a service that never terminates. Its behavior depends on the failure pattern and has no sequential specification.

Differently, given a consensus object *CONS*, when processes invoke *CONS*. propose(), there is a time instant from which they do know that a value has been decided (when considering the previous implementations, this occurs for a process when it reads a non-$\perp$ value from the atomic read/write register *DEC*). There is a single decided value, but that value can be the value proposed by any (correct or faulty) process. To summarize, consensus is a function while a failure detector is not.

17.5 Wait-Free Implementations of the Alpha$_1$ and Alpha$_2$ Abstractions

This section is devoted to the wait-free constructions of the alpha objects defined in Sect. 17.3 and used in Sect. 17.4.

17.5.1 Alpha$_1$ from Atomic Registers

Internal representation The implementation of a (round-free) alpha$_1$ object ($ALPHA1$) rests on two arrays of SWMR atomic registers denoted $AA[1..n]$ and $BB[1..n]$ (where n is the number of processes). $A[i]$ and $B[i]$ can be written only by p_i and read by all the processes. Each array is initialized to $[\perp, \ldots, \perp]$.

The non-atomic operation XX.val_collect() (where XX stands for AA or BB) reads asynchronously all the entries of the array XX and returns the set of values that have been written in X.

```
operation $XX$.val_collect() is
    $aux \leftarrow \emptyset$;
    for $j \in \{1, \ldots, n\}$ do $aux \leftarrow aux \cup \{XX[j]\}$ end for;
    $aux \leftarrow aux \setminus \{\perp\}$;
    return($aux$)
end operation.
```

The algorithm implementing adopt_commit() This algorithm, which is described in Fig. 17.6, uses two communication phases followed by a "return" phase:

- During the first communication phase, a process p_i first deposits into $AA[i]$ the value v_i it proposes (line 1) and then reads asynchronously the values proposed by the other processes (line 2).
- Then p_i executes the second communication phase. If it has seen a single value $v \neq \perp$ in the array $AA[1..n]$ containing proposed values, it writes the pair $\langle single, v\rangle$ into $BB[i]$, otherwise it writes $\langle sevral, v_i\rangle$ (line 3).

  After having informed the other processes of what it has seen, p_i terminates the second communication phase by reading asynchronously the array $BB[1..n]$ (line 4).

```
operation $ALPHA1$.adopt_commit $(v_i)$ is
(1)  $AA[i] \leftarrow v_i$;
(2)  $aa_i \leftarrow AA$.val_collect();
(3)  if $(aa_i = \{v\})$ then $BB[i] \leftarrow \langle single, v\rangle$ else $BB[i] \leftarrow \langle several, v_i\rangle$ end if;
(4)  $bb_i \leftarrow BB$.val_collect();
(5)  case $bb_i = \{\langle single, v\rangle\}$                                  then return($\langle commit, v\rangle$)
(6)       $bb_i = \{\langle single, v\rangle, \langle several, v'\rangle, ...\}$  then return($\langle adopt, v\rangle$)
(7)       $\langle single, v\rangle \notin bb_i$                                 then return($\langle abort, v_i\rangle$)
(8)  end case
end operation.
```

Fig. 17.6 Wait-free implementation of adopt_commit()

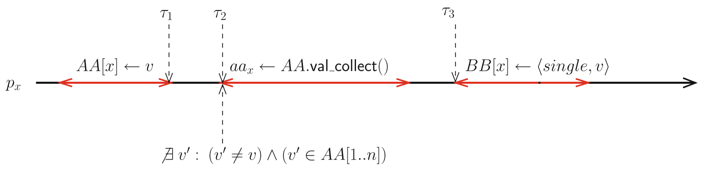
- Finally, p_i computes the final value it returns as the result of its invocation of adopt_commit(v_i):
  - If a single proposed value v was seen by the processes that (to p_i's knowledge) have written into $BB[1..n]$ (those processes have consequently terminated the second communication phase), p_i commits the value v by returning the pair $\langle commit, v\rangle$ (line 5).
  - If the set of pairs read by p_i from $BB[1..n]$ contains several pairs and one of them is $\langle single, v\rangle$, p_i adopts v by returning $\langle adopt, v\rangle$.
  - Otherwise, p_i has not read $\langle single, v\rangle$ from $BB[1..n]$. In that case, it simply returns the pair $\langle abort, v_i\rangle$.

Theorem 81 *The construction described in Fig. 17.6 is a wait-free implementation of an alpha*$_1$ *object from read/write atomic registers.*

Proof The proofs of the termination, input domain, and obligation properties of alpha$_1$ are trivial. Before proving the quasi-agreement property, we prove the following claim.

Claim. $(\langle single, v\rangle \in BB) \Rightarrow (\nexists v' \neq v : \langle single, v'\rangle \in BB)$.
Proof of the claim. Let $\langle single, v\rangle$ be the first pair $\langle single, -\rangle$ written in BB and let p_x be the process that wrote it (due to the atomicity of the registers of the array $BB[1..n]$ such a first write exists).

Due to the sequentiality of p_x we have $\tau_1 < \tau_2 < \tau_3$, where τ_1 is the time at which the statement $AA[x] \leftarrow v$ issued by p_x terminates (line 1), τ_2 is the time instant at



τ_1
τ_2
τ_3
$AA[x] \leftarrow v$
$aa_x \leftarrow AA$.val_collect()
$BB[x] \leftarrow \langle single, v\rangle$
p_x
$\nexists\, v' : (v' \neq v) \wedge (v' \in AA[1..n])$

Fig. 17.7 Timing on the accesses to AA for the proof of the quasi-agreement property

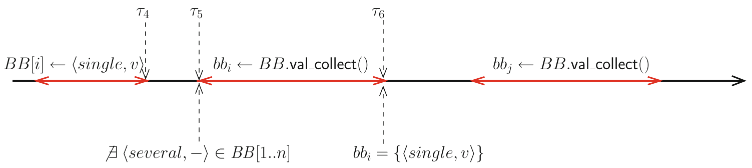

τ_4
τ_5
τ_6
$BB[i] \leftarrow \langle single, v\rangle$
$bb_i \leftarrow BB.\mathsf{val\_collect}()$
$bb_j \leftarrow BB.\mathsf{val\_collect}()$
$\nexists \langle several, -\rangle \in BB[1..n]$
$bb_i = \{\langle single, v\rangle\}$

Fig. 17.8 Timing on the accesses to *BB* for the proof of the quasi-agreement property

which the invocation by p_x of $AA.\mathsf{val\_collect}()$ starts (line 2), and τ_3 is the time at which the statement $BB[x] \leftarrow \langle single, v\rangle$ starts (line 3). See Fig. 17.7.

Let p_y be a process that wrote $v' \neq v$ into the array $AA[1..n]$. As p_x writes $\langle single, v\rangle$ into $BB[x]$, it follows that $aa_x = \{v\}$ (line 3), from which we conclude that, at time τ_2, there is no $v' \neq v$ such that $v' \in AA[1..n]$. It follows that p_y has written v' into $AA[y]$ after time τ_2. Consequently, as $\tau_1 < \tau_2$, p_y executes $aa_y \leftarrow AA.\mathsf{val\_collect}()$ (line 2) and it necessarily reads v from $AA[x]$. It follows that p_y writes $\langle several, v'\rangle$ into $BB[y]$, which concludes the proof of the claim.

The proof of the quasi-agreement property consists in showing that, for any pair of processes that execute line 4, we have $bb_i = \{\langle single, v\rangle\} \Rightarrow \langle single, v\rangle \in bb_j$ (i.e., the lines 5 and 7 are mutually exclusive: if a process executes one of them, no process can execute the other one).

Let p_i be a process that returns $\langle single, v\rangle$ at line 5. It follows that $\langle single, v\rangle$ was written into the array *BB*. It follows from the claim that $\langle single, v\rangle$ is the single pair $\langle single, -\rangle$ written in *BB*. The rest of the proof consists in showing that every process p_j that executes $\mathsf{return}()$ reads $\langle single, v\rangle$ from *BB* at line 4 (from which it follows that it does not execute line 7). The reasoning is similar to the one used in the proof of the claim. See Fig. 17.8 which can be seen as an extension of Fig. 17.7.

If a process p_j writes $\langle several, -\rangle$ into $BB[j]$ (line 3), it does it after time τ_5 (otherwise, p_i could not have $bb_i = \{\langle single, v\rangle\}$ at time τ_6 and executes line 5). As $\tau_4 < \tau_5$, it follows from this observation that, when p_j reads the array $BB[1..n]$ (line 4), it reads $\langle single, v\rangle$ from $BB[i]$ and consequently executes line 6 and returns $\langle adopt, v\rangle$, which concludes the proof of the quasi-agreement property. □

17.5.2 Alpha$_2$ from Regular Registers

This section presents a wait-free implementation of an alpha$_2$ object from SWMR regular registers. As explained in Chap. 11, the main difference between a regular register and an atomic register is the fact that a regular register allows for new/old inversions while an atomic register does not (see Fig. 11.2 and Theorem 43 in Chap. 11). Let us remember that a read of a regular register that is concurrent with one or more write invocations returns the value of the register before these write invocations or a value written by one of these write invocations.

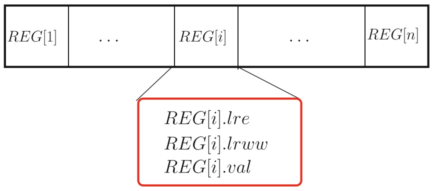


Fig. 17.9 Array of regular registers implementing an alpha$_2$ object

Internal representation of an alpha$_2$ object This representation is made up of an array of regular registers denoted $REG[1..n]$, where n is the number of processes (Fig. 17.9). The register $REG[i]$ can be read by any process, but written only by p_i.

Structure of a regular register Each register $REG[i]$ is made up of three fields (Fig. 17.9). These fields, denoted $REG[i].lre$, $REG[i].lrww$, and $REG[i].val$, are initialized to 0, $-i$, and $\perp$, respectively.

Considering an invocation deposit(r, v) issued by a process p_i, let us remember that the parameter r is a round number, while v is the value that p_i tries to deposit in the corresponding alpha$_2$ object. The three fields of the register $REG[i]$ define its current state with respect to the alpha$_2$ object under construction. Both $REG[i].lre$ and $REG[i].lrww$ store a round number (interpreted as a logical date as we will see below), while $REG[i].val$ stores a proposed value. More precisely, their meaning is the following:

- $REG[i].lre$ stores the number of the *l*ast *r*ound *e*ntered by p_i. It can be seen as the logical date of the last invocation of deposit() issued by p_i.
- $REG[i].lrww$ and $REG[i].val$ constitute a pair of related values: $REG[i].lrww$ stores the number of the *l*ast *r*ound *w*ith a *w*rite of a value v in the field $REG[i].val$. So, $REG[i].lrww$ is the logical date of the last write in $REG[i].val$, which contains the value v that p_i attempts to write into the alpha$_2$ object.

For notational convenience when writing down the algorithm, we consider that each field of $REG[i]$ can be written separately. This is done without loss of generality because, as p_i is the only process that can write into $REG[i]$, it trivially knows its last value. So, as an example, "$REG[i].lre \leftarrow r$" is a shortcut for "$REG[i] \leftarrow \langle r,$ local copy of $REG[i].lrww$, local copy of$REG[i].val\rangle$".

Underlying principles A main difficulty in designing an algorithm implementing the operation deposit() resides in guaranteeing the obligation property (otherwise, returning always $\perp$ would be a correct implementation). So, to guarantee both the quasi-agreement and obligation properties despite concurrent invocations, the idea is to use logical time to timestamp the invocations and answer $\perp$ when the corresponding invocation is not processed according to the total order defined by the timestamps.

The algorithm uses the round numbers as logical dates associated with each invocation (notice that no two invocations have the same dates and the dates used by a

```
operation deposit$(r, v)$ is % This is the text for $p_i$ %
        % [Phase 1] ————————————————————————————————
        % 1.1: $p_i$ first makes public the date (round number) of its last attempt %
(1)     $REG[i].lre \leftarrow r$;
        % 1.2: Then, $p_i$ reads the shared registers to know the other processes progress %
(2)     $reg[1..n] \leftarrow REG[1..n]$; % $p_i$ reads (in any order) the regular registers %
        % 1.3: $p_i$ aborts its attempt if another process started a higher round %
(3)     if $(\exists j :\ reg[j].lre > r)$ then return$(\bot)$ end if;
        % [Phase 2] ————————————————————————————————
        %Then $p_i$ adopts the last value that was deposited in a register %
        % If there is no such value, it adopts its own value $v$ %
(4)     let $value = reg[j].val$ such that $(\forall k :\ reg[j].lrww \geq reg[k].lrww)$;
(5)     if $(value = \bot)$ then $value \leftarrow v$ end if;
        % [Phase 3] ————————————————————————————————
        % 3.1. $p_i$ writes the value it has adopted (together with the current date) %
(6)     $REG[i].\langle lrww, v\rangle \leftarrow \langle r, value\rangle$;
        % 3.2: % $p_i$ reads again the shared registers to know the processes's progress %
(7)     $reg[1..n] \leftarrow REG[1..n]$;
        % 3.3: $p_i$ aborts its attempt if another process started a higher round %
(8)     if $(\exists j :\ reg[j].lre > r)$ then return$(\bot)$ end if;
        % [Phase 4] ————————————————————————————————
        % Otherwise, $value$ is the final value of the alpha object: $p_i$ returns it %
(9)     return$(value)$
end operation.
```

Fig. 17.10 Wait-free construction of an alpha$_2$ object from regular registers

process are increasing). Intuitively, the algorithm aims to ensure that, if there is a "last" invocation of deposit() ("last" with respect to the round numbers, as required by the obligation property), this invocation succeeds in associating a definitive value with the alpha$_2$ object. To that aim, the algorithm manages and uses control information, namely the "date" fields of each shared register (i.e., $REG[i].lre$ and $REG[i].lrww$).

The algorithm implementing the operation deposit() This algorithm is described in Fig. 17.10. Each process p_i has a local array $reg_i[1..n]$ in which p_i stores the last copy of $REG[1..n]$ it has asynchronously read. At the operational level, the previous principles convert into four computation phases. More precisely, when a process p_i invokes deposit(r, v), it executes a sequence of statements which can be decomposed into four phases:

- Phase 1 (lines 1–3):
  - The process p_i first informs the other processes that the date r has been attained (line 1).
  - Then, p_i reads asynchronously the array $REG[1..n]$ to know the "last date" attained by each of the other processes (line 2).
  - If it discovers that it is late (i.e., other processes have invoked deposit() with higher dates), p_i aborts its current attempt and returns $\bot$ (line 3). Let us observe

that this preserves the quasi-agreement property without contradicting the obligation property.

- Phase 2 (lines 4–5). If it is not late, p_i computes a value that it will try to deposit in the alpha$_2$ object. In order not to violate quasi-agreement, it selects the last value ("last" according to the round numbers/logical dates) that has been written in a regular register $REG[j]$. If there is no such value, p_i considers the value v that it wants to deposit in the alpha$_2$ object.
- Phase 3 (lines 6–8):
  - Then, p_i writes into its regular register $REG[i]$ a pair made up of the value (denoted *value*) it has previously computed, together with its date r (line 6).
  - The process p_i then reads again the regular registers to check again if it is late (in which case there are concurrent invocations of deposit() with higher dates). As before, if this is the case, p_i aborts its current attempt and returns $\perp$ (lines 7–8).
- Phase 4 (line 9). Otherwise, as p_i is not late, it has succeeded in depositing v in the alpha$_2$ object, giving it its final value. It consequently returns that value.

Theorem 82 *The construction described in Fig. 17.10 is a wait-free implementation of an alpha$_2$ object from regular registers.*

Proof Proof of the wait-freedom property.
A simple examination of the code of the algorithm shows that it is a wait-free algorithm: if p_i does not crash while executing propose(r, v), it does terminates (at line 3, 8 or 9) whatever the behavior of the other processes.

Proof of the validity property.
Let us observe that a non-$\perp$ value v written in a register $REG[i]$ (line 6) is a value that was previously passed as a parameter in a deposit() invocation (lines 4–6).The validity property follows from this observation and the fact that only $\perp$ or a value written in a register $REG[i]$ can be returned from a deposit() invocation.

Proof of the obligation property.
Let $I =$ deposit($r, -$) be an invocation (by a process p_i) such that all the invocations $I' =$ deposit($r', -$) that have started before I terminates are such that $r' < r$. As $r' < r$, It follows that the predicate $\forall j \neq i : REG[j].lre < r = REG[i].lre$ is true during the whole execution of I. Consequently, I cannot be stopped prematurely and forced to output $\perp$ at line 3 or line 8. It follows that I terminates at line 9 and returns a non-$\perp$ value.

Proof of the quasi-agreement property.
If none or a single invocation of deposit() executes line 9, the quasi-agreement property is trivially satisfied. So, among all the invocations that terminate at line 9 (i.e., that return a non-$\perp$ value), let $I =$ deposit($r, -$) be the invocation with the smallest round number and $I' =$ deposit($r', -$) be any other of these invocations (hence, $r' > r$). (Let us remember that the total ordering on the deposit() invocations defined by their round numbers constitutes the basic idea that underlies the design of

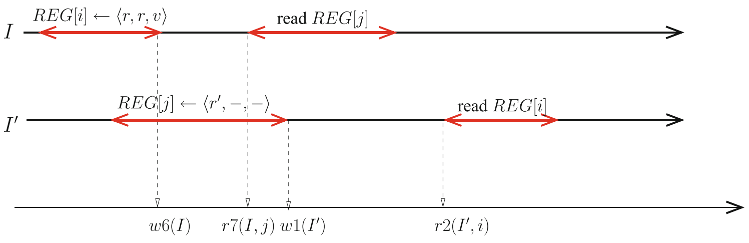

I
$REG[i] \leftarrow \langle r, r, v\rangle$
read $REG[j]$
I'
$REG[j] \leftarrow \langle r', -, -\rangle$
read $REG[i]$
$w6(I)$
$r7(I,j)$
$w1(I')$
$r2(I',i)$

Fig. 17.11 Regular register: read and write ordering

the algorithm.) Let p_i (p_j) be the process that invoked I (I'), and v (v') the returned value. To show that $v = v'$, we use the following time instant definitions (Fig. 17.11):

- Definitions of time instants related to I:
  - Let $w6(I)$ be the time at which I terminates the write of the regular register $REG[i]$ at line 6. We have then $REG[i] = \langle r, r, v\rangle$.
  - Let $r7(I, j)$ be the time at which I starts reading $REG[j]$ at line 7. As p_i is sequential, we have $w6(I) < r7(I, j)$.
- Definitions of time instants related to I':
  - Let $w1(I')$ be the time at which I' terminates the write of the regular register $REG[j]$ at line 1. We then have $REG[j] = \langle r', -, -\rangle$.
  - Let $r2(I', i)$ be the time at which I' starts reading $REG[i]$ at line 2. As p_j is sequential, we have $w1(I') < r2(I', i)$.

Let us first observe that, as I returns a non-$\perp$ value, it successfully passed the test of line 8; i.e., the value it read from $REG[j].lre$ was smaller than r. Moreover, when I' executed line 1, it updated $REG[j].lre$ to $r' > r$. As the register $REG[j]$ is regular, we conclude that I started reading $REG[j]$ before I' finished writing it (otherwise, p_i would have read r' from $REG[j].lre$ and not a value smaller than r). Consequently we have $r7(I, j) < w1(I')$, and by transitivity $w6(I) < r7(I, j) < w1(I') < r2(I', i)$. This is illustrated in Fig. 17.11.

It follows that, when I' reads $REG[i]$ at line 2, it obtains $\langle x, x, -\rangle$ with $x \geq r$ (this is because, after I, p_i has possibly executed other invocations with higher round numbers). Moreover, as I' does not return $\perp$ at line 3, when it read $REG[1..n]$ at line 2 it saw no $REG[k]$ such that $REG[k].lre > r'$. This means that, when I' determines a value val at line 4, it obtains v' from some register $REG[k]$ (the value in $REG[k].val$) such that $\forall \ell : r' > REG[k].lrww \geq REG[\ell].lrww$, and we have $REG[k].lrww \geq REG[i].lrww = r$. Let I'' be the invocation that deposited v' in $REG[k].val$. If $REG[k].lrww = r$, we then have $i = k$ and I'' is I (this is because r can be generated only by p_i). Consequently $v' = v$. Otherwise, the invocation I'' by p_k deposited v' into $REG[k].val$ at line 6, with a corresponding round number r'' such that $r < REG[k].lrww = r'' < r'$. Observing that only lines 1–4 executed by I' are relevant in the previous reasoning, we can reuse the same reasoning replacing the pair of invocations (I, I') with the pair (I, I''). So, either I'' obtained v' deposited by I and then $v' = v$, or I'' obtained v' from another invocation I''', and so on. When considering the sequence of invocations defined by the round numbers, the number of invocations between I and I' is finite (there are at most $r' - r + 1$ invocations in this sequence). It follows that the chain of invocations conveying v' to I' is finite, and can only start with the invocation I that deposited v in the corresponding register. (Thank you Pierre de Fermat!) Consequently, $v' = v$. $\square$

17.6 Wait-Free Implementations of the Alpha$_2$ Abstraction from Shared Disks

17.6.1 Alpha$_2$ from Unreliable Read/Write Disks

Motivation An aim of this section is to show how a construction suited to a new context (unreliable read/write disks) can be derived by applying simple transformations to a construction designed for a simpler context (read/write regular registers).

Commodity disks Advances in hardware technology enable a new approach for storage sharing, where clients can access disks directly over a *storage area network* (SAN). The disks are directly attached to high-speed networks that are accessible to clients. A client can access raw disk data (mediated by disk controllers with limited memory and CPU capabilities). These disks (usually called *commodity disks* or *network attached disks*, NAD) are cheaper than computers and are consequently attractive for achieving fault-tolerance. This has motivated the design of disk-based consensus algorithms.

Shared disk model The "shared memory" is now made up of m disks, each disk containing n blocks (one per process). $DISK\_BK[i, d]$ denotes the block associated with p_i on disk d, $1 \leq d \leq m$ (accordingly, $DISK\_BK[-, d]$ denotes the whole disk d, see Fig. 17.12).

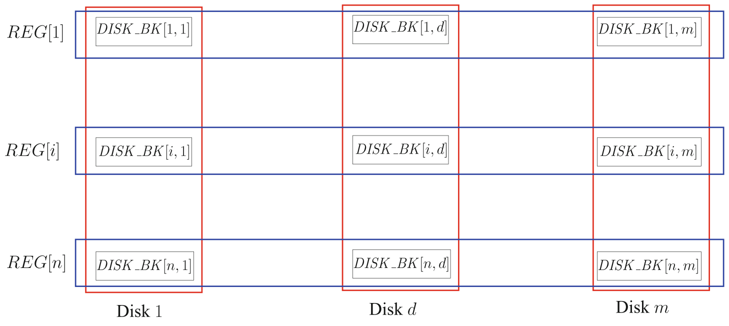


Fig. 17.12 Replicating and distributing $REG[1..n]$ on the m disks

A disk block *DISK_BK*$[i, d]$ can be accessed by a read or a write operation. These operations are atomic (this means that all the read and write operations on a disk block can be totally ordered in a consistent way). A disk can crash. When the disk d crashes, all its blocks *DISK_BK*$[i, d]$ $(1 \leq i \leq n)$ become inaccessible, and the corresponding read and write operations return $\perp$ or never return (as seen in Sect. 15.1, an invocation that returns a value or $\perp$ is *responsive* while an invocation that never returns is *non-responsive*). A disk that crashes in a run is *faulty* in that run, otherwise it is *correct*. It is assumed that a majority of disks are correct.

Underlying principles of the construction When we consider the algorithm described in Fig. 17.10 that constructs an alpha object, we can observe that, if each register *REG*$[i]$ is implemented by a reliable read/write disk, the algorithm still works. So, the idea consists in the following:

- First, replicating each register *REG*$[i]$ on each disk d $(1 \leq d \leq m)$ in order to make that register fault-tolerant (*REG*$[i]$ is consequently implemented by m copies, namely the disk blocks *DISK_BK*$[i, 1]$, ..., *DISK_BK*$[i, m]$). This replication scheme is described in Fig. 17.13. For any pair (i, d), $1 \leq i \leq n$, $1 \leq d \leq m$, the disk block *DISK_BK*$[i, d]$ is consequently composed of three fields denoted *lre*, $lrww$, and *val* which define a triple *triple* with the same meaning as in Sect. 17.5.2. Each triple is initialized to $\langle 0, -i, \perp \rangle$.
- Then, applying classic quorum-based replication techniques to the disks, where a disk quorum is any majority set of disks. The assumption on the majority of correct disks guarantees that there is always a live quorum (this ensures termination), and two disk quorums always intersect (this ensures safety).

  A *quorum* is a set of processes. Quorums are defined in such a way that any two quorums intersect. This intersection property is used to propagate information know by the processes in a given quorum to the processes of another quorum, the propagation being done by the process(es) that belong to the intersection of the quorums.

Building a reliable regular register from unreliable disks As just indicated, the disk-based implementation of alpha$_2$ consists in using the quorum-based replication technique to translate the read and write operations on a "virtual" object *REG*$[i]$ into its disk access counterpart on the m copies *DISK_BK*$[i, 1]$, ..., *DISK_BK*$[i, m]$.

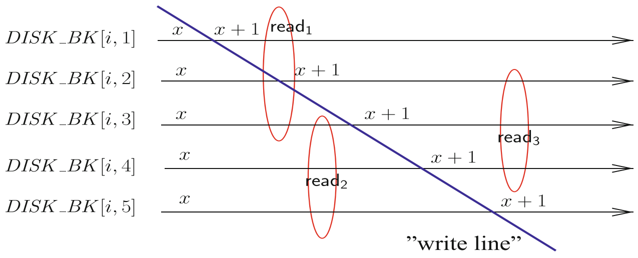


Fig. 17.13 Implementing an SWMR regular register from unreliable read/write disks

To that aim, each time it has to write a new triple into the virtual object $REG[i]$, p_i associates a new sequence number sn with that triple $triple$ and writes the pair $\langle triple, sn\rangle$ into the corresponding block $DISK\_BK[i, d]$ of each of the m disks ($1 \leq d \leq m$). The write invocation terminates when the pair has been written into a majority of disks.

Similarly, a read of the virtual object $REG[i]$ is translated into m read operations, each one reading the corresponding block $DISK\_BK[i, d]$ on disk d ($1 \leq d \leq m$). The read terminates when a pair was received from a majority of the disks; the triple with the greatest sequence number is then delivered as the result of the read of the virtual object $REG[i]$. Let us observe that, due to the "majority of correct disks" assumption, every read or write operation on a virtual object $REG[i]$ terminates.

It is easy to see that a read of the virtual object $REG[i]$ that is not concurrent with a write of it obtains the last triple written in $REG[i]$ (or $\perp$ if $REG[i]$ has never be written). For the read operations of $REG[i]$ that are concurrent with a write in $REG[i]$, let us consider Fig. 17.13, which considers five disk blocks: $DISK\_BK[i, 1], \ldots, DISK\_BK[i, 5]$. A write of a virtual object $REG[i]$ by p_i is represented by a "write line", the meaning of which is the following: the point where the "write line" crosses the time line of a disk is the time at which that disk executes the write of the corresponding $\langle triple, sn\rangle$ pair. As the system is asynchronous, these physical writes can occur as indicated in the figure, whatever their invocation times. When considering the figure, x is the sequence number associated with the value of $REG[i]$ in each disk block $DISK\_BK[i, d]$, $1 \leq d \leq m = 5$, before the write; $x + 1$ is consequently the sequence number after the write operation. (Let us observe that, in general, due to asynchrony and the fact that an operation terminates when a new value was written into a majority of disks, it is possible that two disks $DISK\_BK[i, d]$ and $DISK\_BK[i, d']$ do not have the same sequence number.)

The figure considers also three reads of $REG[i]$: each is represented by an ellipsis and obtains the <triple, sequence number> pair of the disks contained in the corresponding ellipsis (e.g., read_2 obtains the current pairs of the disk blocks $DISK\_BK[i, 3]$, $DISK\_BK[i, 4]$, and $DISK\_BK[i, 5]$). As we can see, each read obtains pairs from a majority of disks (let us notice that this is the best that can be done as, from the invoking process point of view, all the other disks may have crashed). read_1 and read_2 are concurrent with the write into the virtual object $REG[i]$, and read_1 obtains the new triple (whose sequence number is $x + 1$), while read_2 obtains the old triple (whose sequence number is x). This is a new/old inversion. As we have seen, the regular register definition allows such new/old inversions.

Let us finally notice that a write on the virtual object $REG[i]$ such that p_i crashes during that write operation can leave the disk blocks $DISK\_BK[i, 1], \ldots, DISK\_BK[i, m]$ in a state where the pair $\langle$new value of $REG[i]$, associated sequence number$\rangle$ has not been written into a majority of disks. Considering that a write during which the corresponding process crashes never terminates, this remains consistent with the definition of a regular register. As shown by Fig. 17.13, all future reads will then be concurrent with that write and each of them can consequently return the old or the new value of $REG[i]$.

The algorithm implementing the operation deposit() When we look at the basic construction described in Fig. 17.10 with the "sequence number" notion in mind, we can see that the fields *REG*[*i*].*lre* and *REG*[*i*].$lrww$ of each virtual object *REG*[*i*] actually play a sequence number role: the first for the writes of *REG*[*i*] issued at line 1, and the second for the writes of *REG*[*i*] issued at line 6. These fields of each virtual register *REG*[*i*] can consequently be used as sequence numbers.

On another side, as far as disk accesses are concerned, we can factor out the writing of *REG*[*i*] at line 1 and the reading of *REG*[1..*n*] at line 2. This means that we can issue, for each disk d, the writing of *DISK_BK*[i, d] and the reading of *DISK_BK*[$1, d$], ..., *DISK_BK*[n, d], and wait until these operations have been executed on a majority of disks. The same factorization can be done for the writing of *REG*[*i*] at line 6 and the reading of *REG*[1..*n*] at line 7 of Fig. 17.10.

The resulting construction based on unreliable disks is described in Fig. 17.14. The variable *reg*[*i*] is a local variable where p_i stores the last value of *REG*[*i*] (while there

```
operation deposit($r, v$) is % This is the text for $p_i$ %
        % $\boxed{\text{Phase 1}}$ ————————————————————————————————
        % 1.1. $p_i$ first makes public the date of its last attempt %
        % 1.2. and reads the shared disks to know the other processes progress %
(1)     $reg[i].lre \leftarrow r$;
(2)     concurrently for each disk $d$ : $1 \le d \le m$ do
(3)                 issue write $reg[i]$ into $DISK\_BK[i, d]$;
(4)                 issue read $DISK\_BK[1, d]$, ..., $DISK\_BK[m, d]$ end for;
(5)     wait (this was successfully done at a majority of disks);
(6)     let $read\_blocks$ = the set of triples $block[j, d].\langle lre.lrww, value\rangle$ that have been read;
        % 1.3: $p_i$ aborts its attempt if another process started a higher round %
(7)     if ($\exists\, block[j, d] \in read\_blocks$ : $block[j, d].lre > r$) then return($\perp$) end if;
        % $\boxed{\text{Phase 2}}$ ————————————————————————————————
        % Then $p_i$ adopts the last value that was written %
        % If there is no such value, it adopts its own value $v$ %
(8)     let $value = block[j, d].val$ such that $(block[j, d] \in read\_blocks) \wedge$
                 $(\forall\, k, d' : block[k, d'] \in read\_blocks : block[j, d].lrww \ge block[k, d'].lrww)$;
(9)     if $(value = \perp)$ then $value \leftarrow v$ end if;
        % $\boxed{\text{Phase 3}}$ ————————————————————————————————
        % 3.1. $p_i$ writes the value it has adopted (together with the current date $r$) %
        % 3.2. and reads the shared disks to know the other processes progress %
(10)    $reg[i] \leftarrow \langle r, r, value\rangle$;
(11)    concurrently for each disk $d$ : $1 \le d \le m$ do
(12)                issue write $reg[i]$ into $DISK\_BK[i, d]$;
(13)                issue read $DISK\_BK[1, d]$, ..., $DISK\_BK[m, d]$ end for;
(14)    wait (this was successfully done at a majority of disks);
(15)    let $read\_blocks$ = the set of triples $block[j, d].\langle lre, -, -\rangle$ that have been read;
        % 3.3. $p_i$ aborts its attempt if another process started a higher round %
(16)    if ($\exists\, block[j, d] \in read\_blocks$ : $block[j, d].lre > r$) then return($\perp$) end if;
        % $\boxed{\text{Phase 4}}$ ————————————————————————————————
        % Otherwise, $value$ is the final value the alpha object: $p_i$ returns it %
(17)    return($value$)
operation.
```

Fig. 17.14 Wait-free construction of an alpha$_2$ object from unreliable read/write disks

is no local array $reg[1..n]$, we keep the array-like notation $reg[i]$ for homogeneity and notational convenience). The algorithm tolerates any number of process crashes, and up to $\lfloor (m-1)/2 \rfloor$ disk crashes. It is wait-free as a process can always progress and terminate its deposit() invocation despite the crash of any number of processes.

This algorithm can be easily improved. As an example, when p_i receives at line 4 a triple $block[j,d]$ such that $block[j,d].lre > r$, it can abort the current attempt and return $\perp$ without waiting for triples from a majority of disks. (The same improvement can be done at line 13.)

17.6.2 Alpha$_2$ from Active Disks

An *active disk* is a disk that can atomically execute a few operations more sophisticated than read or write. (As an example, active disks have been implemented that provide their users with an atomic create() operation that atomically creates a new file object and updates the corresponding file directory.)

We consider here an active disk, denoted *ACT_DISK*, that is made up of three fields: *ACT_DISK.lre*, *ACT_DISK.lrww*, and *ACT_DISK.val*. This disk can be atomically accessed by the two operations described in Fig. 17.15. The first operation, denoted write_round + read(), takes a round number r as input parameter. It updates the field *ACT_DISK.lre* to max(*ACT_DISK.lre*, r) and returns the triple contained in *ACT_DISK*. The second operation, denoted cond_write + read_round(), takes a triple as input parameter. It is a conditional write that returns a round number.

The algorithm implementing the operation deposit() This algorithm is described in Fig. 17.16. It is a simple adaptation of the algorithm described in Fig. 17.10 to the active disk context in which the array $REG[1..n]$ of base regular registers is replaced by a more sophisticated object, namely the active disk. Each process p_i manages two local variables: act_disk_i, which is a local variable that contains the last value of the active disk read by p_i, and rr_i, which is used to contain a round number. It is easy to see that this construction is wait-free.

```
(1) operation write_round + read_triple($r$) is
(2)    $AC\_DISK.lre \leftarrow \max(AC\_DISK.lre, r)$;
(3)    return($AC\_DISK$)
end operation.

(4) operation write_val + read($triple$) is
(5)    if ( $(triple.lre \geq AC\_DISK.lre)$ $\wedge$ $(triple.lrww > AC\_DISK.lrww)$ )
(6)                    then $AC\_DISK \leftarrow triple$ end if;
(7)    return($AC\_DISK.lre$)
end operation.
```

Fig. 17.15 The operations of an active disk

```
operation deposit$(r, v)$ is % This is the text for $p_i$ %
        % Phase 1 ———————————————————————————————
        % 1.1. $p_i$ first makes public the date of its last attempt %
        % 1.2. and reads the active disk to know the other processes progress %
(1)     $act\_disk_i \leftarrow ACT\_DISK$.write_round + read_triple$(r)$;
        % 1.3: $p_i$ aborts its attempt if another process started a higher round %
(2)     if $(act\_disk_i.lre > r)$ then return$(\perp)$ end if;
        % Phase 2 ———————————————————————————————
        % Then $p_i$ adopts the last value that was deposited %
        % If there is no such value, it adopts its own value $v$ %
(3)     if $(ac\_disk_i.val \neq \perp)$ then $value \leftarrow ac\_disk_i.val$ else $value \leftarrow v$ end if;
        % Phase 3 ———————————————————————————————
        % 3.1. $p_i$ writes the value it has adopted (together with the current date) %
        % 3.2. and reads the active disk to know the other processes progress %
(4)     $rr \leftarrow ACT\_DISK$.write_val + read$(\langle r, r, value\rangle)$;
        % 3.3: $p_i$ aborts its attempt if another process started a higher round %
(5)     if $(rr > r)$ then return$(\perp)$ end if;
        % Phase 4 ———————————————————————————————
        % Otherwise, $value$ is the final value the alpha object: $p_i$ returns it %
(6)     return$(value)$
end operation.
```

Fig. 17.16 Wait-free construction of an alpha$_2$ object from an active disk

Unreliable active disks The previous construction assumes that the underlying active disk is reliable. An interesting issue is to build a reliable virtual active disk from base unreliable active disks. Unreliability means here that an active disk can crash but does not corrupt its values. After it has crashed (if it ever crashes) a disk no longer executes the operations that are applied to it. Before crashing it executes them atomically. A disk that crashes during a run is said to be *faulty* with respect to that run; otherwise, it is *correct*. Let us assume there are *m* active disks. It is possible to build a correct active disk.

Using the quorum-based approach described in Sect. 17.6.1, it is possible to build a correct active disk from a set of *m* active disks in which no more than $m/2$ may crash.

17.7 Implementing Ω

This section is devoted to the implementation of Ω in an asynchronous system where up to *t* processes may crash, $1 \leq t \leq n-1$, and the processes communicate by accessing atomic read/write registers.

As (a) Ω allows wait-free implementation of a consensus object despite asynchrony and any number of process crash failures and (b) a consensus object cannot be implemented from read/write registers only, it follows that Ω cannot be implemented from read/write registers only. This means that the underlying system has to be enriched (or, equivalently, restricted) in order for Ω to be built. This is captured by additional behavioral assumptions that have to be at the same time "weak enough" (to be practically nearly always satisfied) and "strong enough" (to allow Ω to be implemented).

The section presents first timing assumptions which are particularly weak and then shows that these timing assumptions are strong enough to implement Ω in any execution where they are satisfied. These assumptions and the corresponding construction of Ω are due to A. Fernández, E. Jiménez, M. Raynal and G. Trédan (2007, 2010).

17.7.1 The Additional Timing Assumption EWB

Each process is assumed to be equipped with a timer $timer_i$, and any two timers measure time durations the same way. The assumption is denoted *EWB* (for *e*ventually *w*ell *b*ehaved). The intuition that underlies *EWB* is that (a) on the one side the system must be "synchronous" enough for a process to have a chance to be elected and (b) on the other side the other processes must be able to recognize it.

EWB is consequently made up of two parts: EWB_1, which is on the existence of a process whose behavior satisfies some synchrony assumption, and EWB_2, which is on the timers of other processes. Moreover, EWB_1 and EWB_2 have to be complementary assumptions that have to fit each other.

Critical registers Some SWMR atomic registers are critical, while other are not. A *critical* register is an atomic register on which some timing constraint is imposed by *EWB* on the single process that is allowed to write this register. This attribute allows one to restrict the set of registers involved in the *EWB* assumption.

The assumption EWB_1 This assumption restricts the asynchronous behavior of a single process. It is defined as follows:

> EWB_1: There are a time τ_{EWB_1}, a bound Δ, and a correct process p_ℓ (τ_{EWB_1}, Δ, and p_ℓ may be never explicitly known) such that, after τ_{EWB_1}, any two consecutive write accesses issued by p_ℓ to its critical register are completed in at most Δ time units.

This property means that, after some arbitrary (but finite) time, the speed of p_ℓ is lower-bounded; i.e., its behavior is partially synchronous (let us notice that, while there is a lower bound, no upper bound is required on the speed of p_ℓ, except the fact that it is not $+\infty$). In the following we say "p_ℓ satisfies EWB_1" to say that p_ℓ is a process that satisfies this assumption.

The assumption EWB_2 This assumption, which is on timers, is based on the following timing property. Let a timer be *eventually well behaved* if there is a time τ_{EWB_2} after which, whatever the finite duration δ and the time $\tau' \geq \tau_{EWB_2}$ at which it is set to δ, that timer expires at some finite time τ'' such that $\tau'' \geq \tau' + \delta$. It is important to notice that an eventually well behaved timer is such that, after some time, it never expires too early.

This definition allows a timer to expire at arbitrary times (i.e., times that are unrelated to the duration it was set to) during an arbitrary but finite time, after which it behaves correctly.

Let t be the maximal number of processes that may crash in an execution and f the number of processes that crash. Hence, $0 \leq f \leq t \leq n-1$. Let us notice that, while t is a constant known by the processes, the value of f depends on each execution and is not known by the processes.

Given the previous definitions, the assumption EWB_2 is defined as follows:

> EWB_2: The timers of $(t-f)$ correct processes, different from the process p_ℓ that satisfies EWB_1, are eventually well behaved.

When we consider *EWB*, it is important to notice that any process (except one, which is constrained by a lower bound on its speed) can behave in a fully asynchronous way. Moreover, the local clocks used to implement the timers are not required to be synchronized.

Let us also observe that the timers of up to $n-(t-f)$ correct processes can behave arbitrarily. It follows that, in the executions where $f = t$, the timers of all the correct processes can behave arbitrarily. It follows from these observations that the timing assumption *EWB* is particularly weak.

In the following we say "p_x is involved in EWB_2" to say that p_x is a correct process that has an eventually well-behaved timer.

17.7.2 An *EWB*-Based Implementation of Ω

Principle of the construction The construction is based on a simple idea: a process p_i elects the process that is the least suspected to have crashed. So, each time a process p_i suspects p_j because it has not observed progress from p_j during some duration (defined by the latest timeout value used to set its timer), it increases a suspicion counter (denoted $SUSPICIONS[i, j]$).

It is possible that, because its timer does not behave correctly, a process p_i erroneously suspects a process p_k, despite the fact that p_k did make some progress (this progress being made visible thanks to assumption EWB_1 if p_k satisfies that assumption). So, when it has to compute its current leader, p_i does not consider all the suspicions. For each process p_k, it takes into account only the $(t+1)$ smallest values among the n counters $SUSPICIONS[1, k], \ldots, SUSPICIONS[n, k]$. As we will see, due to EWB_2, this allows it to eliminate the erroneous suspicions.

As several processes can be equally suspected, the processes use the function $\min(X)$, where X is a set of pairs $\langle a, i\rangle$ (i is a process identity and a the number of times p_i was suspected); $\min(X)$ returns the smallest pair, according to lexicographical ordering, in the set X (let us remember that $\langle a, i\rangle < \langle b, j\rangle$ if and only if $(a < b) \vee (a = b \wedge i < j)$).

Internal representation of Ω To implement Ω, the processes cooperate by reading and writing two arrays of SWMR atomic registers:

- $PROGRESS[1..n]$ is an array of SWMR atomic registers that contain positive integers. It is initialized to $[1, \dots, 1]$. Only p_i can write $PROGRESS[i]$, and it does so regularly to inform the other processes that it is alive. Each $PROGRESS[i]$ register is a *critical* register.
- $SUSPICIONS[1..n, 1..n]$ is an array of (non-critical) SWMR atomic registers that contain positive integers (each entry being initialized to 1). The vector $SUSPICIONS[i, 1..n]$ can be written only by p_i. $SUSPICIONS[i, j] = x$ means that p_i has suspected $x - 1$ times the process p_j to have crashed.

Local variables Each process p_i manages the following local variables:

- $progress_i$ is used by p_i to measure its own progress and update accordingly $PROGRESS[i]$.
- $last_i[1..n]$ is an array such that $last_i[k]$ contains the greatest value of $PROGRESS[k]$ known by p_i.
- $suspicions_i[1..n]$ is an array such that $suspicions_i[k]$ contains the number of times p_i has suspected p_k to have crashed.
- $progress\_k_i$, $timeout_i$, and $susp_i[1..n]$ are auxiliary local variables used by p_i to locally memorize relevant global values.

Behavior of a process The behavior of a process to implement Ω is described in Fig. 17.17. It is decomposed into three tasks whose executions are mutually exclusive.

The first task ($T1$) defines the way the current leader is determined. For each process p_k, p_i first computes the number of relevant suspicions that concern p_k. As already mentioned, those are the $(t + 1)$ smallest values in the vector $SUSPICIONS[1..n, k]$ (line 3). The current leader is then defined as the process currently the least suspected, considering only the relevant suspicions (line 5). There is no constraint on the speed or the periodicity with which the task T_1 is repeatedly executed.

The second task ($T2$) is a simple repetitive task whose aim is to increase $PROGRESS[i]$ in order to inform the other processes that p_i has not crashed (line 8). There is no constraint on the speed or the periodicity with which the task $T2$ is repeatedly executed for all the processes except one, namely the process involved in the assumption EWB_1.

The third task ($T3$) is executed each time the timer of p_i expires. It is where p_i possibly suspects the other processes and where it sets its timer ($timer_i$) as follows:

1. Suspicion management part (lines 11–18). For each process p_k ($k \neq i$), p_i first reads $PROGRESS[k]$ to see p_k's current progress. If there is no progress since the last reading of $PROGRESS[k]$, p_i suspects once more p_k to have crashed, and consequently increases $SUSPICIONS[i, k]$.

2. Timer setting part (lines 19–23). Then, p_i resets its timer to an appropriate timeout value. That value is computed from the current relevant suspicions. Let us observe that this timeout value increases when these suspicions increase. Let us also remark that, if after some time the number of relevant suspicions no longer increases, *timeout*$_i$ keeps forever the same value.

As we can see, the construction is relatively simple. It uses n^2+n atomic registers. (As, for any i, *SUSPICIONS*$[i, i]$ is always equal to 1, it is possible to use the diagonal of that matrix to store the array *PROGRESS*$[1..n]$.)

17.7.3 Proof of the Construction

Let us consider an execution of the construction described in Fig. 17.17 in which the assumptions EWB_1 and EWB_2 defined in Sect. 17.7.1 are satisfied. This section

```
task $T1$:
(1)  repeat forever:
(2)     for each $k \in \{1, ..., n\}$ do
(3)        let $susp_i[k] = \Sigma$ of the $(t+1)$ smallest values in the vector $SUSPICIONS[1..n, k]$
(4)     end for;
(5)     $leader_i \leftarrow \ell$ where $\ell$ is such that $(-, \ell) = \min(\{(susp_i[k], k)\}_{1 \le k \le n})$
(6)  end repeat.

task $T2$:
(7)  repeat forever
(8)        $progress_i \leftarrow progress_i + 1$; $PROGRESS[i] \leftarrow progress_i$
(9)  end repeat.

task $T3$:
(10) when $timer_i$ expires do
(11)    for each $k \in \{1, ..., n\} \setminus \{i\}$ do
(12)       $progress\_k_i \leftarrow PROGRESS[k]$;
(13)       if $(progress\_k_i \neq last_i[k])$
(14)          then $last_i[k]$                $\leftarrow progress\_k_i$
(15)          else $suspicions_i[k]$          $\leftarrow suspicions_i[k] + 1$;
(16)               $SUSPICIONS[i, k]$  $\leftarrow suspicions_i[k]$
(17)       end if
(18)    end for;
(19)    for each $k \in \{1, ..., n\}$ do
(20)       let $susp_i[k] = \Sigma$ of the $(t+1)$ smallest values in the vector $SUSPICIONS[1..n, k]$
(21)    end for;
(22)    let $timeout_i = min(\{susp_i[k]\}_{1 \le k \le n})$;
(23)    set $timer_i$ to $timeout_i$
(24) end when.
```

Fig. 17.17 A t-resilient construction of Ω (code for p_i)

shows that an eventual leader is elected in that run. The proof is decomposed into several lemmas.

Lemma 41 *Let p_i be a faulty process. For any p_j, $SUSPICIONS[i, j]$ is bounded.*

Proof Let us first observe that the vector $SUSPICIONS[i, 1..n]$ is updated only by p_i. The proof follows immediately from the fact that, after it has crashed, a process no longer updates atomic registers. □

Lemma 42 *Let p_i and p_j be a correct and a faulty process, respectively. $SUSPICIONS[i, j]$ grows forever.*

Proof After a process p_j has crashed, it no longer increases the value of $PROGRESS[j]$, and consequently, due to the update of line 14, there is a finite time after which the test of line 13 remains forever false for any correct process p_i. It follows that $SUSPICIONS[i, j]$ increases without bound at line 15. □

Lemma 43 *Let p_i be a correct process involved in the assumption EWB_2 (i.e., its timer is eventually well behaved) and let us assume that, after some point in time, $timer_i$ is always set to a value $\Delta' > \Delta$. Let p_j be a correct process that satisfies the assumption EWB_1. Then, $SUSPICIONS[i, j]$ is bounded.*

Proof As p_i is involved in EWB_2, there is a time τ_{EWB_2} such that $timer_i$ never expires before $\tau + \delta$ if it was set to δ at time τ, with $\tau \geq \tau_{EWB_2}$. Similarly, as p_j satisfies EWB_1, there are a bound Δ and a time τ_{EWB_1} after which two consecutive write operations issued by p_j into $PROGRESS[j]$ are separated by at most Δ time units (let us recall that $PROGRESS[j]$ is the only critical variable written by p_j).

Let τ_Δ be the time after which $timeout_i$ takes only values $\Delta' > \Delta$, and let $\tau = \max(\tau_\Delta, \tau_{EWB_1}, \tau_{EWB_2})$. As after time τ_Δ any two consecutive write operations into $PROGRESS[j]$ issued by p_j are separated by at most Δ time units, while any two reading of $PROGRESS[j]$ by p_i are separated by at least Δ' time units, it follows that there is a finite time $\tau' \geq \tau$ after which we always have $PROGRESS[j] \neq last_i[j]$ when evaluated by p_i (line 12). Hence, after τ', the shared variable $SUSPICIONS[i, j]$ is no longer increased, which completes the proof of the lemma. □

Definition 6 Given a process p_k, let $sk_1(\tau) \leq sk_2(\tau) \leq \cdots \leq sk_{t+1}(\tau)$ denote the $(t + 1)$ smallest values among the n values in the vector $SUSPICIONS[1..n, k]$ at time τ. Let $M_k(\tau)$ denote $sk_1(\tau) + sk_2(\tau) + \cdots + sk_{t+1}(\tau)$.

Definition 7 Let S denote the set containing the f faulty processes plus the $(t - f)$ processes involved in the assumption EWB_2 (whose timers are eventually well behaved). Then, for each process $p_k \notin S$, let S_k denote the set $S \cup \{p_k\}$. (Let us notice that $|S_k| = t + 1$.)

Lemma 44 *At any time τ, there is a process $p_i \in S_k$ such that the predicate $SUSPICIONS[i, k] \geq sk_{t+1}(\tau)$ is satisfied.*

Proof Let $K(\tau)$ be the set of the $(t + 1)$ processes p_x such that, at time τ, $SUSPICIONS[x, k] \leq sk_{t+1}(\tau)$. We consider two cases:

1. $S_k = K(\tau)$. Then, taking p_i as the "last" process of S_k such that *SUSPICIONS* $[i, k] = sk_{t+1}(\tau)$ proves the lemma.
2. $S_k \neq K(\tau)$. In this case, let us take p_i as a process in $S_k \setminus K(\tau)$. As $p_i \notin K(\tau)$, it follows from the definition of $K(\tau)$ that *SUSPICIONS*$[i, k] \geq sk_{t+1}(\tau)$, and the lemma follows. □

Definition 8 Let $M_x = \max(\{M_x(\tau)_{\tau \geq 0}\})$. If there is no such value (then $M_x(\tau)$ grows forever according to τ). let $M_x = +\infty$. Let B be the set of processes p_x such that M_x is bounded.

Lemma 45 *If the assumption EWB_1 is satisfied, then $B \neq \emptyset$.*

Proof Let p_k be a process that satisfies EWB_1. We show that M_k is bounded. Due to Lemma 44, at any time τ, there is a process $p_{j(\tau)} \in S_k$ such that we have *SUSPICIONS*$[j(\tau), k](\tau) \geq sk_{t+1}(\tau)$ (where *SUSPICIONS*$[j(\tau), k](\tau)$ denotes the value of the corresponding variable at time τ). It follows that $M_k(\tau)$ is upper bounded by $(t + 1) \times$ *SUSPICIONS*$[j(\tau), k](\tau)$. So, the proof amounts to showing that, after some time, for any $j \in S_k$, *SUSPICIONS*$[j, k]$ remains bounded. Let us consider any process $p_j \in S_k$ after the time at which the f faulty processes have crashed. There are three cases:

1. $p_j = p_k$. In this case we always have *SUSPICIONS*$[j, k] = 1$.
2. p_j is a faulty process of S_k. In this case, the fact that *SUSPICIONS*$[j, k]$ is bounded follows directly from Lemma 41.
3. p_j is a process of S_k that is one of the $(t - f)$ correct processes involved in the assumption EWB_2. The fact that *SUSPICIONS*$[j, k]$ is bounded is then an immediate consequence of Lemma 43. □

Lemma 46 *B does not contain faulty processes.*

Proof Let p_j be a faulty process. Observe that, for each τ, the set of processes whose values in the vector *SUSPICIONS*$[1..n, j]$ are added to compute $M_j(\tau)$ contains at least one correct process. Due to Lemma 42, for each correct process p_i, *SUSPICIONS*$[i, j]$ increases forever, and hence so does M_j, which proves the lemma. □

Lemma 47 *Let p_i be a correct process. There is a time after which any read of $leader_i$ by p_i returns the identity of a (correct) process of B.*

Proof The lemma follows from (a) lines 2–5 and (b) the fact that B is not empty (Lemma 45) and contains only correct processes (Lemma 46). □

Definition 9 Let $(M_\ell, \ell) = \min(\{(M_x, x) | p_x \in B\})$.

Lemma 48 *There is a single process p_ℓ, and it is a correct process.*

Proof The lemma follows directly from the following observations: B does not contain faulty processes (Lemma 46), it is not empty (Lemma 45), and no two processes have the same identity. □

Theorem 83 *There is a time after which the local variable $leader_i$ of all the correct processes remain forever equal to the same identity, which is the identity of a correct process.*

Proof The theorem follows from Lemma 47 and Lemma 48. □

17.7.4 Discussion

On the process that is elected The proof of the construction relies on the assumption EWB_1 to guarantee that at least one correct process can be elected; i.e., the set B is not empty (Lemma 45) and does not contain faulty processes (Lemma 46). This does not mean that the elected process is a process that satisfies the assumption EWB_1. There are cases where it can be another process.

To see when this can happen, let us consider two correct processes p_i and p_j such that p_i does not satisfy EWB_2 (its timer is never well behaved) and p_j does not satisfy EWB_1 (it never behaves synchronously with respect to its critical register $PROGRESS[j]$). (A re-reading of the statement of Lemma 43 will make the following description easier to understand.) Despite the fact that (1) p_i is not synchronous with respect to a process that satisfies EWB_1, and can consequently suspect these processes infinitely often, and (2) p_j is not synchronous with respect to a process that satisfies EWB_2 (and can consequently be suspected infinitely often by such processes), it is still possible that p_i and p_j behave synchronously with respect to each other in such a way that p_i never suspects p_j. If this happens $SUSPICIONS[i, j]$ remains bounded, and it is possible that the value M_j not only remains bounded but becomes the smallest value in the set B. It this occurs, p_j is elected as the common leader.

Of course, there are runs in which the previous scenario does not occur. That is why the protocol has to rely on EWB_1 in order to guarantee that the set B is never empty.

On the timeout values It is important to notice that the timeout values are determined from the least suspected processes. Moreover, after the common leader (say p_ℓ) was elected, any timeout value is set to M_ℓ. It follows that, given any run, be it finite or infinite, the timeout values are always bounded with respect to that run (two executions can have different bounds).

17.8 Summary

This chapter has investigated the construction of a consensus object in an asynchronous system where processes communicate through read/write registers and any number of processes may crash. To that end, an abstraction related to information on failure provided to processes (failure detector Ω) and three abstractions (denoted alpha_1, alpha_2, and alpha_3) have been presented. Ω provides processes with the additional computability power that allows their invocations of the consensus object to terminate, while an alpha abstraction (which can be wait-free implemented from read/write registers only) allows one to guarantee the safety consensus property (validity and agreement). Wait-free constructions of consensus objects based on these abstractions have been described and proved correct.

The chapter has also described several read/write-based implementations of the alpha objects and presented a weak timing assumption (denoted *EWB*) that allows Ω to be built in all executions where it is satisfied.

17.9 Bibliographic Notes

- The failure detector abstraction was introduced by T. Chandra and S. Toueg [67]. The failure detector Ω was introduced by T. Chandra, V. Hadzilacos and S. Toueg [68], who proved in that paper that it captures the weakest information on failure to solve the consensus problem.
- The first use of a failure detector to solve the consensus problem in a shared memory system appeared in [197].

  In addition to the Ω-based consensus algorithms described in this chapter, other Ω-based consensus algorithms for shared memory systems can be found in [84, 240]. Ω-based consensus algorithms suited to message-passing systems can be found in several papers (e.g., [213, 127, 129]).

  Relations between failure detectors and wait-freedom are studied in [218]. An introduction to failure detectors can be found in [235].

  An extension of failure detectors to bounded lifetime failure detectors was introduced in [104]. Such an extension allows a leader to be elected for a finite period of time (and not “forever”).

- The impossibility of solving the consensus problem in the presence of asynchrony and process crashes was first proved by M. Fischer, N.A. Lynch, and M.S. Paterson in the context of message-passing systems [102]. The first proof suited to shared memory systems is due to M. Loui and H. Abu-Amara [199].
- The alpha_1 (adopt-commit) object and its wait-free implementation are due to E. Gafni [105]. This object was used in [275] to address agreement problems. A message-passing version was developed independently in [212].

- The alpha$_2$ object is due to L. Lamport [192], who introduced it in the context of message-passing systems (Paxos algorithm).
- A technique similar to the one used to implement an alpha$_2$ object was used in timestamp-based transaction systems. A timestamp is associated with each transaction, and a transaction is aborted when it accesses data that has already been accessed by another transaction with a higher timestamp (an aborted transaction has to be re-issued with a higher timestamp [50]).
- The implementation of an alpha$_2$ object based on unreliable disks is due E. Gafni and L. Lamport [109].
- The consensus algorithm based on a store-collect object presented in Fig. 17.5 (Sect. 17.4.3) is due to M. Raynal and J. Stainer [239]. It originates from an algorithm described in [240] (which itself was inspired by an algorithm presented by C. Delporte and H. Fauconnier in [84]).
- The power of read/write disks encountered in storage area networks is investigated in [15].
- The implementation of an alpha$_2$ object based on active disks is due to G. Chockler and D. Malkhi [75]. Generalization to Byzantine failures can be found in [1].
- The notion of an indulgent algorithm is due to R. Guerraoui [120]. Its underlying theory is developed in [126], and its application to message-passing systems is investigated in [92, 127, 129, 247, 274].
- The implementation of Ω described in Sect. 17.7 is due to A. Fernández, E. Jiménez, M. Raynal, and G. Trédan [100]. An improvement of this construction in which, after some finite time, only the eventual leader writes the shared memory is presented in [101]: (See also [99].)
- The implementation of Ω in crash-prone asynchronous message-passing systems was addressed in several papers (e.g., [75, 99, 128]).

17.10 Exercises and Problems

1. Design a consensus algorithm based on alpha$_1$ objects in which any subset of processes can participate. (Hint: consider the failure detector Ω_X introduced in Sect. 5.3.1.)

   Solution in [242].

2. A k-set object is a weakened form of a consensus object in which each process decides a value, and at most k different values can be decided. (Hence, consensus is 1-set agreement.)

   Generalize the construction of the alpha$_1$ and alpha$_2$ objects so that they can be used to solve k-set agreement.

```
operation $CS$.collect() is
(1)     $res_i \leftarrow \emptyset$;
(2)     for $j \in \{1...,n\}$ do
(3)        $reg \leftarrow REG[j]$;
(4)        if $(reg \neq \perp)$ then $res_i \leftarrow res_i \cup \{reg\}$ end if
(5)     end for;
(6)     return$(res_i)$
end operation.

operation $CS$.deposit$(v)$ is
(7)     $aux_i \leftarrow CS$.collect();
(8)     if $(aux_i = \emptyset)$
(9)        then $REG[i] \leftarrow v$; return$(v)$
(10)       else let $w$ be any value in $aux_i$; return$(w)$
(11)    end if
end operation.
```

Fig. 17.18 The operations collect() and deposit() on a closing set object (code for process p_i)

Solution in [242] for alpha$_2$ based on regular registers.

3. Prove that the consensus algorithm based on a store-collect object (alpha$_3$) and Ω described in Sect. 17.3.3 (Fig. 17.5) is correct.

   Solution in [240].

4. Construct an alpha$_2$ object from a set of m active disks such that at most $m/2$ may crash.

5. Simulate the behavior of an alpha$_2$ object in a message-passing system in which a majority of processes remain correct, and use it to design an Ω-based message-passing consensus object.

   Solution in [129] (which can be seen as a version of the Paxos algorithm [192] adapted to asynchronous reliable communication).

6. Modify and extend the construction of Ω described in Sect. 17.7 in order to obtain an implementation in which, after some finite time, a single process (the eventual leader) writes the shared memory.

   Solution in [101].

7. Let us consider the following alpha$_4$ object, called a *closing set*. Such an object *CS* is a special type of set defined by two operations. The first is *CS*.collect(), which returns a set $S \subseteq CS$ which is not empty if values have already been deposited in *CS*. The second is *CS*.deposit(v), which tries to deposit the value v into *CS*. This deposit is successful if $CS = \emptyset$ from the invoking process point of view. If values have already been deposited, the closing set *CS* is closed (no more values can be deposited). Each invocation of *CS*.deposit() returns a value that was deposited in *CS*.

Differently from the alpha$_1$ (adopt-commit) and alpha$_2$ objects, a closing set has no sequential specification and, consequently, is not an atomic object. Moreover, similarly to alpha$_1$, it is a round-free object (rounds do not appear in its definition).

A wait-free implementation of a closing set is described in Fig. 17.18. This implementation uses an array of SWMR atomic registers $REG[1..n]$ initialized to $[\perp, \ldots, \perp]$. The aim of $REG[i]$ is to contain the value deposited by p_i into the closing set CS.

Considering an unbounded sequence of closing sets objects, $CS[0]$, $CS[1]$, $CS[2], \ldots$, design an Ω-based wait-free construction of a consensus object and prove that it is correct.

Solution in [240].

Afterword

What Was the Aim of This Book

The practice of sequential computing has greatly benefited from the results of the theory of sequential computing that were captured in the study of formal languages and automata theory. Everyone knows what can be computed (computability) and what can be computed efficiently (complexity). All these results constitute the foundations of sequential computing, which, thanks to them, has become a *science*. These theoretical results and algorithmic principles have been described in a lot of books from which students can learn basic results, algorithms, and principles of sequential computing (e.g., [79, 85, 114, 151, 176, 203, 211, 258] to cite a few).

Synchronization is coming back, but is it the same? While books do exist for traditional synchronization (e.g., [27, 58]), very few books present in a comprehensive way the important results that have been discovered in the past 20 years.[1] Hence, even if it describes a lot of algorithms implementing concurrent objects, the aim of this book is not to be a catalog of algorithms. Its ambition is not only to present synchronization algorithms but also to introduce the reader to the theory that underlies the implementation of concurrent objects in the presence of asynchrony and process crashes.

To summarize, thanks to appropriate curricula (and good associated books), students now have a good background in the theory and practice of sequential computing. An aim of this book is to try to provide them with an equivalent background when they have to face the net effect of asynchrony and failures in the context of shared memory systems.[2] Hence, all the concepts, principles, and mechanisms presented in the book aim at attaining this target.

[1] The book by Gadi Taubenfeld [262] and the book by Maurice Herlihy and Nir Shavit [146] are two such books.

[2] For message-passing systems, the reader can consult [40, 60, 115, 176, 201, 236, 237, 248].

Technology is what makes everyday life easier. Science is what allows us to transcend it, and capture the deep nature of the objects we are manipulating. To that end, it provides us with the right concepts to master and understand what we are doing. When considering synchronization and concurrent objects encountered in computing science, an ambition of this book is to be a step in this direction.

Most Important Concepts, Notions, and Mechanisms Presented in This Book

Chapter 1: Competition, concurrent object, cooperation, deadlock-freedom, invariant, liveness, lock object, multiprocess program, mutual exclusion, safety, sequential process, starvation-freedom, synchronization.

Chapter 2: Atomic read/write register, lock object, mutual exclusion, safe read/write register, specialized hardware primitive (test&set, fetch&add, compare&swap).

Chapter 3: Declarative synchronization, imperative synchronization, lock-based implementation, monitor, path expression, predicate transfer, semaphore.

Chapter 4: Atomicity, legal history, history, linearizability, locality property, partial operation, sequential consistency, sequential history, serializability, total operation.

Chapter 5: Contention manager, implementation boosting, mutex-freedom, obstruction-freedom, non-blocking, process crash, progress condition, wait-freedom.

Chapter 6: Abortable object, binary consensus, concurrent set object, contention-sensitive implementation, double-ended queue, hybrid (static versus dynamic) implementation, LL/SC primitive operations, linearization point, non-blocking to starvation-freedom.

Chapter 7: Adaptive implementation, fast store-collect, favorable circumstances, infinitely many processes, store-collect object, weak counter.

Chapter 8: Atomic snapshot object, immediate snapshot object, infinitely many processes, one-shot object, read/write system, recursive algorithm.

Chapter 9: Adaptive algorithm, immediate snapshot object, grid of splitters, recursive algorithm, renaming object (one-shot versus long-lived).

Chapter 10: Abort, atomic execution unit, commit, deferred updates, incremental read (snapshot), lock, multi-version, opacity, read invisibility, read/modify/write, software transactional memory, speculative execution, transaction (read-only, write-only, update), virtual world consistency.

Chapter 11: Atomic/regular/safe register, binary versus multi-valued register, bounded versus unbounded construction, register construction, sequence number, SWSR/SWMR/MWSR/MWMR, single-reader versus multi-reader, single-writer versus multi-writer, timestamp.

Chapter 12: Atomic bit, bounded construction, linearizability, safe bit, switch, wait-free construction.

Chapter 13: Atomic bit, atomic register construction, collision-freedom, finite state automaton, linearizability, pure/impure buffer, safe buffer, switch, SWSR register, wait-freedom.

Chapter 14: Atomic object, binary consensus, bounded construction, consensus object, deterministic versus non-deterministic object, linearizability, multi-valued consensus, sequential specification, total operation, total order, universal construction, universal object, wait-freedom.

Chapter 15: Arbitrary failure, atomic read/write register, bounded versus unbounded implementation, consensus object, crash failure, graceful degradation, omission failure, responsive versus non-responsive failure, self-implementation, *t*-tolerant implementation, wait-freedom.

Chapter 16: Atomic object, bivalent configuration, consensus object, consensus number, consensus hierarchy, indistinguishability, monovalent configuration, valence, wait-freedom.

Chapter 17: Active disk, adopt-commit, alpha abstraction, atomic register, compare&swap, consensus object, indulgence, omega abstraction, regular register, round-based abstraction, shared disk, store-collect, timing assumption.

How to Use This Book

This section presents courses on synchronization and concurrent objects which can benefit from the concepts, algorithms and principles presented in this book. The first course consists of one semester course for the last year of the undergraduate level, while the two one-semester courses are more appropriate for the graduate level.

- Undergraduate students.

  A one-semester course could first focus on Part I devoted to mutual exclusion. (The reader can notice this part of the book has plenty of exercises.)

  Then, the course could address (a) the underlying theory, namely Part II devoted to the formalization of the atomicity concept (in order for students to have a clear view of the foundations on what they have learned), and (b) the notion of mutex-freedom and associated progress conditions introduced in Chap. 5. Examples taken from Chaps. 7–9 can be used to illustrate these notions.

Finally, Part IV devoted to software transactional memory can be used to introduce students to a new approach that addresses the design of multiprocess programs.

- Graduate students.

  Assuming that the content of the one-semester course for undergraduate students is known and mastered, a first one-semester course for graduate students could first study the notion of hybrid implementation (Chap. 6) and then investigate the power of read/write registers to wait-free implement concurrent objects. This is what is addressed in detail in Chaps. 5–9.

  Then, the course could survey Part V which describes various constructions of atomic read/write registers from safe registers. A selection of algorithms presented in Chaps. 11–13 can be used to that end.

  A second one-semester course should be devoted to computability in the presence of failures, which is the topic addressed in Part VI of the book. The notions of a universal construction (Chap. 14), its reliability (Chap. 15), the consensus number hierarchy (Chap. 16), and the construction of consensus objects (Chap. 17) are fundamental notions to master both the practice and the theory of synchronization in the presence of failures.

Of course, this book can also be used by engineers and researchers who work on shared memory multiprocessor systems, or multicore architectures, and are interested in (a) *what* can be done in the presence of asynchrony and failures, and (b) *how* it has to be done.

A Look at Other Books

This section briefly presents a few other books which address synchronization in shared memory systems.

- Books on message-passing and shared memory:
  - The book by H. Attiya and J. Welch [41], the book by A. Kshemkalyani and M. Singhal [181], and the book by N. Lynch [201] consider both the message-passing model and the shared memory model.

    More precisely, Part IIA of Lynch's book is devoted to the asynchronous shared memory model. The algorithms are described in the input/output automata model and formally proved using this formalism. In addition to mutual exclusion, this part of the book visits also resource allocation.

    Chapter 5 of Attiya and Welch's book, and Chap. 9 of Kshemkalyani and Singhal's book, are fully devoted to the mutual exclusion problem in shared memory systems. Both books present also constructions of atomic read/write registers from weaker registers.

  – The construction of a shared memory on top of an asynchronous message-passing system where processes may crash is presented in the previous three books, and in Chap. 4 of a book by C. Cachin, R. Guerraoui, and L. Rodrigues [60]. Such constructions constitute a main part of a book by M. Raynal [236]. (More generally, this last book is entirely devoted to failure-prone message-passing systems and the use of failure detectors to circumvent impossibility results encountered in these systems.)

- Recent books entirely devoted to shared memory systems.
  – Taubenfeld's book [262] covers a lot of results motivated by problems which arise when building systems. The book presents a lot of algorithms mainly devoted to process coordination (mutual exclusion, resource allocation, barrier synchronization, etc.). It also addresses timing-based synchronization algorithms. An impressive number of results related to complexity bounds for shared memory synchronization are also presented.
  – Herlihy and Shavit's book [146] is made up of two main parts. The first addresses base principles of process synchronization and concurrent objects. The second part is practice-oriented. It presents a huge number of algorithms implementing concurrent objects which are encountered in a lot of applications (e.g., linked lists, concurrent stacks, skip lists, priority queues). All the algorithms are described in Java.

... Marco Polo describes a bridge, stone by stone.
'But which is the stone that supports the bridge?' Kublai Khan asks.
'The bridge is not supported by one stone or another', Marco answers,
'but by the line of the arch that they form'.
In *Invisible Cities* (1963), Italo Calvino (1923–1985).

... Y colorin colorado, este cuento se ha acabado.
Anonymous, Spanish culture.

Bibliography

1. I. Abraham, G.V. Chockler, I. Keidar, D. Malkhi, Byzantine disk Paxos, optimal resilience with Byzantine shared memory. *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, St. John's, 2004 (ACM Press, New York, 2004), pp. 226–235
2. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
3. Y. Afek, G. Brown, M. Merritt, Lazy caching. ACM Trans. Program. Lang. Syst. **15**(1), 182–205 (1993)
4. Y. Afek, D. Dauber, D. Touitou, Wait-free made fast. *Proceedings of the 27th ACM Symposium on Theory of Computing (STOC'00)*, Portland, 2000 (ACM Press, New York, 2000), pp. 538–547
5. Y. Afek, E. Gafni, A. Morisson, Common2 extended to stacks and unbounded concurrency. Distrib. Comput. **20**(4), 239–252 (2007)
6. Y. Afek, E. Gafni, S. Rajsbaum, M. Raynal, C. Travers, The k-simultaneous consensus problem. Distrib. Comput. **22**(3), 185–195 (2010)
7. Y. Afek, I. Gamzu, I. Levy, M. Merritt, G. Taubenfeld, Group renaming. *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, Luxor, 2008. LNCS, vol. 5401 (Springer, Berlin, 2006), pp. 58–72
8. Y. Afek, D. Greenberg, M. Merritt, G. Taubenfeld, Computing with faulty shared objects. J. ACM **42**(6), 1231–1274 (1995)
9. Y. Afek, M. Merritt, Fast wait-free $(2k-1)$-renaming. *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, Atlanta, 1999 (ACM Press, New York, 1999), pp. 105–112
10. Y. Afek, M. Merritt, G. Taubenfeld, The power of multi-objects. Inf. Comput. **153**, 213–222 (1999)
11. Y. Afek, M. Merritt, G. Taubenfeld, D. Touitou, Disentangling multi-object operations. *Proceedings of the 16th International ACM Symposium on Principles of Distributed Computing (PODC'97)*, Santa Barbara, 1997 (ACM Press, New York, 1997), pp. 262–272
12. Y. Afek, G. Stupp, D. Touitou, Long-lived adaptive collect with applications. *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science Computing (FOCS'99)*, New York, 1999 (IEEE Computer Press, New York, 1999), pp. 262–272
13. Y. Afek, E. Weisberger, H. Weisman, A completeness theorem for a class of synchronization objects. *Proceedings of the 12th International ACM Symposium on Principles of Distributed Computing (PODC'93)*, Ithaca, 1993 (ACM Press, New York, 1993), pp. 159–168
14. M.K. Aguilera, A pleasant stroll through the land of infinitely many creatures. ACM SIGACT News, Distrib. Comput. Column **35**(2), 36–59 (2004)

15. M.K. Aguilera, B. Englert, E. Gafni, On using network attached disks as shared memory. *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC'03)*, Boston, 2003 (ACM Press, New York, 2003), pp. 315–324
16. M.K. Aguilera, S. Frolund, V. Hadzilacos, S.L. Horn, S. Toueg, Abortable and query-abortable objects and their efficient implementation. *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, Portland, 2007 (ACM Press, New York, 2007), pp. 23–32
17. M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, Ph.W. Hutto, Causal memory: definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (1995)
18. D. Alistarh, J. Aspnes, S. Gilbert, R. Guerraoui, The complexity of renaming. *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2011)*, Palm Springs, 2011 (IEEE Press, New York, 2011), pp. 718–727
19. D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, R. Guerraoui, Fast randomized test-and-set and renaming. *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, Cambridge, 2010. LNCS, vol. 6343 (Springer, Heidelberg, 2010), pp. 94–108
20. B. Alpern, F.B. Schneider, Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
21. J.H. Anderson, Composite registers. *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC'90)*, Quebec City, 1990 (ACM Press, New York, 1990), pp. 15–29
22. J.H. Anderson, Multi-writer composite registers. Distrib. Comput. **7**(4), 175–195 (1994)
23. J.H. Anderson, Y.J. Kim, Adaptive local exclusion with local spinning. *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, Toledo, 2000. LNCS, vol. 1914 (Springer, Heidelberg, 2000), pp. 29–43
24. J.H. Anderson, Y.J. Kim, T. Herman, Shared memory mutual exclusion: major research trends since 1986. Distrib. Comput. **16**, 75–110 (2003)
25. J. Anderson, M. Moir, Universal constructions for multi-object operations. *Proceedings of the 14th International ACM Symposium on Principles of Distributed Computing (PODC'95)*, Ottawa, 1995 (ACM Press, New York, 1995), pp. 184–195
26. J. Anderson, M. Moir, Universal constructions for large objects. IEEE Trans. Parallel Distrib. Syst. **10**(12), 1317–1332 (1999)
27. G.R. Andrews, *Concurrent Programming, Principles and Practice* (Benjamin/Cumming, Redwood City, 1993), 637 pp
28. A.A. Aravind, Yet another simple solution to the concurrent programming control problem. IEEE Tran. Parallel Distrib. Syst. **22**(6), 1056–1063 (2011)
29. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, R. Reischuk, Renaming in an asynchronous environment. J. ACM **37**(3), 524–548 (1990)
30. H. Attiya, E. Dagan, Improved implementations of universal binary operations. J. ACM **48**(5), 1013–1037 (2001)
31. H. Attiya, F. Ellen, P. Fatourou, The complexity of updating snapshot objects. J. Parallel Distrib. Comput. **71**(12), 1570–1577 (2010)
32. H. Attiya, A. Fouren, Polynomial and adaptive long-lived $(2p-1)$-renaming. *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, Toledo, 2000. LNCS, vol. 1914 (Springer, Heidelberg, 2000), pp. 149–163
33. H. Attiya, A. Fouren, Adaptive and efficient algorithms for lattice agreement and renaming. SIAM J. Comput. **31**(2), 642–664 (2001)
34. H. Attiya, A. Fouren, Algorithms adapting to point contention. J. ACM **50**(4), 444–468 (2003)
35. H. Attiya, A. Fouren, E. Gafni, An adaptive collect algorithm with applications. Distrib. Comput. **15**(2), 87–96 (2002)
36. H. Attiya, R. Guerraoui, E. Ruppert, Partial snapshot objects. *Proceedings of the 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, Munich, 2008 (ACM Press, New York, 2008), pp. 336–343
37. H. Attiya, E. Hillel, Highly concurrent multi-word synchronization. *Proceedings of the 9th International Conference on Distributed Computing and Networking (ICDCN'08)*, Kolkata, 2008. LNCS, vol. 4904 (Springer, Heidelberg, 2008), pp. 112–123

38. H. Attiya, E. Hillel, Single-version STM can be multi-version permissive. *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN'11)*, Bangalore, 2011. LNCS, vol. 6522 (Springer, Heidelberg, 2011), pp. 83–94
39. H. Attiya, O. Rachmann, Atomic snapshots in $O(n \log n)$ operations. SIAM J. Comput. **27**(2), 319–340 (1998)
40. H. Attiya, J.L. Welch, Sequential consistency versus linearizability. ACM Trans. Comput. Syst. **12**(2), 91–122 (1994)
41. H. Attiya, J.L. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd edn. (Wiley-Interscience, New York, 2004), 414 pp. ISBN 0-471-45324-2
42. H. Avni, N. Shavit, Maintaining consistent transactional states without a global clock. *Proceedings of the 15th Colloquium on Structural Information and Communication Complexity (SIROCCO'08)*, Villars-sur-Ollon, 2008. LNCS, vol. 5058 (Springer, Heidelberg, 2008), pp. 121–140
43. B. Awerbuch, L.M. Kirousis, E. Kranakis, P. Vitányi, A proof technique for register atomicity. *Proceedings of the 8th International Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, LNCS, vol. 338 (Springer, 1988), pp. 286–303
44. T. Axford, Concurrent programming. Fundamental techniques for real-time and parallel software design. *Wiley Series in Parallel Computing* (Wiley, New York, 1989), 250 pp
45. Ö. Babaoglu, K. Marzullo, Consistent global states of distributed systems: fundamental concepts and mechanisms. *Chapter 4 in Distributed Systems*. Frontier Series (ACM Press, New York, 1993), pp. 55–93
46. Y. Bar-David, G. Taubenfeld, Automatic discovery of mutual exclusion algorithms. *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, Sorrento, 2003. LNCS, vol. 2648 (Springer, Heidelberg, 2003), pp. 136–150
47. G. Barnes, A method for implementing lock-free shared data structures. *Proceedings of the 5th International ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)*, Venlo, 1993 (ACM Press, New York, 1993), pp. 261–270
48. A.J. Bernstein, Predicate transfer and timeout in message passing systems. Inf. Process. Lett. **24**(1), 43–52 (1987)
49. A.J. Bernstein, Ph.L. Lewis, *Concurrency in Programming and Database Systems* (John and Bartlett, Boston, 1993), 548 pp
50. Ph.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems* (Addison-Wesley, Reading, 1987), 370 pp
51. Ph.A. Bernstein, D.W. Shipman, W.S. Wong, Formal aspects of serializability in database concurrency control. IEEE Trans. Softw. Eng. **SE-5**(3), 203–216 (1979)
52. B. Bloom, Constructing two-writer atomic register. IEEE Trans. Comput. **37**, 1506–1514 (1988)
53. E. Borowsky, E. Gafni, Immediate atomic snapshots and fast renaming. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, Ithaca, 1993, pp. 41–51
54. E. Borowsky, E. Gafni, A simple algorithmically reasoned characterization of wait-free computations. *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, Santa Barbara, 1997 (ACM Press, New York, 1997), pp. 189–198
55. E. Borowsky, E. Gafni, Y. Afek, On processor coordination with asynchronous hardware. Consensus power makes (some) sense. *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, Los Angeles, 1994 (ACM Press, New York, 1994), pp. 363–372
56. P. Brinch Hansen, Shared classes. In *Operating Systems Principles (Section 7.2)* (Prentice Hall, Englewood Cliffs, 1973), pp. 226–232
57. P. Brinch Hansen, *The Architecture of Concurrent Programs* (Prentice Hall, Englewood Cliffs, 1977), 317 pp
58. P. Brinch Hansen (ed.), *The Origin of Concurrent Programming* (Springer, Heidelberg, 2002), 534 pp

59. J.E. Burns, G.L. Peterson, Constructing multireader atomic values from non-atomic values. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC'87)*, Vancouver, 1987 (ACM Press, New York, 1987), pp. 222–231
60. Ch. Cachin, R. Guerraoui, L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming* (Springer, New York, 2011), 320 pp
61. J. Cachopo, A. Rito-Silva, Versioned boxes as the basis for transactional memory. Sci. Comput. Program. **63**(2), 172–175 (2006)
62. R.H. Campbell, Predicates path expressions. *Proceedings of the 6th ACM Symposium on Principles of Programming Languages (POPL'79)*, San Antonio, 1979 (ACM Press, New York, 1979), pp. 226–236
63. R.H. Campbell, N. Haberman, The specification of process synchronization by path expressions. *Proceedings of the International Conference on Operating Systems*, LNCS, vol. 16 (Springer, Berlin, 1974), pp. 89–102
64. R.H. Campbell, R.B. Kolstad, Path expressions in Pascal. *Proceedings of the 4th International Conference on Software Engineering (ICSE'79)*, Munich, 1979 (ACM Press, New York, 1979), pp. 212–219
65. A. Castañeda, S. Rajsbaum, New combinatorial topology upper and lower bounds for renaming: the lower bound. Distrib. Comput. **22**(5), 287–301 (2010)
66. A. Castañeda, S. Rajsbaum, M. Raynal, The renaming problem in shared memory systems: an introduction. Elsevier Comput. Sci. Rev. **5**, 229–251 (2011)
67. T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
68. T. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
69. K.M. Chandy, J. Misra, *Parallel Program Design* (Addison-Wesley, Reading, 1988), 516 pp
70. B. Charron-Bost, R. Cori, A. Petit, Introduction à l'algorithmique des objets partagés. RAIRO Informatique Théorique et Applications **31**(2), 97–148 (1997)
71. S. Chaudhuri, More choices allow more faults: set consensus problems in totally asynchronous systems. Inf. Comput. **105**(1), 132–158 (1993)
72. S. Chaudhuri, M.J. Kosa, J. Welch, One-write algorithms for multivalued regular and atomic registers. Acta Inform. **37**, 161–192 (2000)
73. S. Chaudhuri, J. Welch, Bounds on the cost of multivalued registers implementations. SIAM J. Comput. **23**(2), 335–354 (1994)
74. G.V. Chockler, D. Malkhi, Active disk Paxos with infinitely many processes. Distrib. Comput. **18**(1), 73–84 (2005)
75. G.V. Chockler, D. Malkhi, Light-weight leases for storage-centric coordination. Int. J. Parallel Prog. **34**(2), 143–170 (2006)
76. B. Chor, A. Israeli, M. Li, On processor coordination with asynchronous hardware. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC'87)*, Vancouver, 1987 (ACM Press, New York, 1987), pp. 86–97
77. Ph. Chuong, F. Ellen, V. Ramachandran, A universal construction for wait-free transaction friendly data structures. *Proceedings of the 22nd International ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, Santorini, 2010 (ACM Press, New York, 2010), pp. 335–344
78. R, Colvin, L. Groves, V. Luchangco, M. Moir, Formal verification of a lazy concurrent list-based set algorithm. *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, Seattle, 2006. LNCS, vol. 4144 (Springer, Heidelberg, 2006), pp. 475–488
79. Th.M. Cormen, Ch.E. Leiserson, R.L. Rivest, *Introduction to Algorithms* (The MIT Press, Cambridge, 1998), 1028 pp
80. P.J. Courtois, F. Heymans, D.L. Parnas, Concurrent control with "readers" and "writers". Commun. ACM **14**(5), 667–668 (1971)
81. T. Crain, V. Gramoli, M. Raynal, A speculation-friendly binary search tree. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, New Orleans, 2012 (ACM Press, New York, 2012), pp. 161–170

82. T. Crain, D. Imbs, M. Raynal, Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'11)*, Melbourne, 2011. LNCS, vol. 7016 (Springer, Berlin, 2011), pp. 245–258
83. T. Crain, D. Imbs, M. Raynal, Towards a universal construction for transaction-based multiprocess programs. *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN'12)*, Hong Kong, 2012. LNCS, vol. 7129, (Springer, Berlin, 2012), pp. 61–75
84. C. Delporte-Gallet, H. Fauconnier, Two consensus algorithms with atomic registers and failure detector *Ω*. *Proceedings of the 10th International Conference on Distributed Computing and Networking (ICDCN'09)*, Hyderabad, 2009. LNCS, vol. 5408 (Springer, Heidelberg, 2009), pp. 251–262
85. P.J. Denning, J.B. Dennis, J.E. Qualitz, *Machines Languages and Computation* (Prentice Hall, Englewood Cliffs, 1978), 612 pp
86. D. Dice, V.J. Marathe, N. Shavit, Lock cohorting: a general technique for designing NUMA locks. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, New Orleans, 2012 (ACM Press, New York, 2012), pp. 247–256
87. D. Dice, O. Shalev, N. Shavit, Transactional locking II. *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, Stockholm, 2006. LNCS, vol. 4167 (Springer, Heidelberg, 2006), pp. 194–208
88. E.W. Dijkstra, Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569 (1965)
89. E.W. Dijkstra, Cooperating Sequential Processes. In *Programming Languages*, ed. by F. Genuys (Academic Press, New York, 1968), pp. 43–112
90. E.W. Dijkstra, Hierarchical ordering of sequential processes. Acta Inform. **1**(1), 115–138 (1971)
91. A.B. Downey, *The Little Book of Semaphores*, 2nd edn, version 2.1.2. (Green Tea Press, Virginia, 2005), 291 pp. http://www.greenteapress.com/semaphores/downey05semaphores.pdf
92. P. Dutta, R. Guerraoui, Fast indulgent consensus with zero degradation. *Proceedings of the 4th European Dependable Computing Conference (EDCC'02)*, Toulouse, 2002. LNCS, vol. 2485 (Springer, Heidelberg, 2002), pp. 191–208
93. F. Ellen, How hard is it to take a snapshot? *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'05)*, Liptovský Ján, 2005. LNCS, vol. 3381 (Springer, Heidelberg, 2005), pp. 28–37
94. B. Englert, E. Gafni, Fast collect in the absence of contention. *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, 2002 (IEEE Press, New York, 2002), pp. 537–543
95. P. Fatourou, N.D. Kallimanis, The red-blue adaptive universal construction. *Proceedings of the 22nd International Symposium on Distributed Computing (DISC'09)*, Elche, 2009. LNCS, vol. 5805 (Springer, Berlin, 2009), pp. 127–141
96. P. Fatourou, N.D. Kallimanis, A highly-efficient wait-free universal construction. *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*, San Jose, 2011 (ACM Press, New York, 2011), pp. 325–334
97. A. Fekete, N. Lynch, M. Merritt, W. Weihl, *Atomic Transactions* (Morgan Kaufmann, San Mateo, 1994)
98. P. Felber, Ch. Fetzer, R. Guerraoui, T. Harris, Transactions are coming back, but are they the same? ACM SIGACT News, Distrib. Comput. Column **39**(1), 48–58 (2008)
99. A. Fernández, E. Jiménez, M. Raynal, Electing an eventual leader in an asynchronous shared memory system. *Proceedings of the 37th International IEEE Conference on Dependable Systems and Networks (DSN'07)*, Edinburgh, 2007 (IEEE Computer Society Press, New York, 2007), pp. 399–408
100. A. Fernández, E. Jiménez, M. Raynal, G. Trédan, A timing assumption and a *t*-resilient protocol for implementing an eventual leader in asynchronous shared memory systems. *Proceedings of the 10th International IEEE Symposium on Objects and Component-*

Oriented Real-Time Computing (ISORC 2007), Santorini Island, May 2007 (IEEE Society Computer Press, New York, 2007), pp. 71–78

101. A. Fernández, E. Jiménez, M. Raynal, G. Trédan, A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. Algorithmica **56**(4), 550–576 (2010)
102. M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
103. A. Fouren, Exponential examples of two renaming algorithms. *Technion Tech Report*, 1999. http://www.cs.technion.ac.il/hagit/pubs/expo.ps.gz
104. R. Friedman, A. Mostéfaoui, M. Raynal, Asynchronous bounded lifetime failure detectors. Inf. Process. Lett. **94**(2), 85–91 (2005)
105. E. Gafni, Round-by-round fault detectors: unifying synchrony and asynchrony. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, Puerto Vallarta, 1998 (ACM Press, New York, 1998), pp. 143–152
106. E. Gafni, Group solvability. *Proceedings of the 18th International Symposium on Distributed Computing (DISC'04)*, Amsterdam, 2004. LNCS, vol. 3274 (Springer, Heidelberg, 2004), pp. 30–40
107. E. Gafni, Renaming with k-set consensus: an optimal algorithm in $n+k-1$ slots. *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS'06)*, Bordeaux, 2006. LNCS, vol. 4305 (Springer, Heidelberg, 2006), pp. 36–44
108. E. Gafni, R. Guerraoui, Generalizing universality. *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR'11)*, Aachen, 2011. LNCS, vol. 6901 (Springer, Berlin, 2011), pp. 17–27
109. E. Gafni, L. Lamport, Disk Paxos. Distrib. Comput. **16**(1), 1–20 (2003)
110. E. Gafni, M. Merritt, G. Taubenfeld, The concurrency hierarchy, and algorithms for unbounded concurrency. *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, Newport, 2001, pp. 161–169
111. G. Gafni, A. Mostéfaoui, M. Raynal, C. Travers, From adaptive renaming to set agreement. Theoret. Comput. Sci. **410**(14–15), 1328–1335 (2009)
112. E. Gafni, S. Rajsbaum, Recursion in distributed computing. *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, New York, 2010. LNCS, vol. 6366 (Springer, Heidelberg, 2010), pp. 362–376
113. E. Gafni, M. Raynal, C. Travers, Test&set, adaptive renaming and set agreement: a guided visit to asynchronous computability. *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, Beijing, 2007 (IEEE Computer Society Press, New York, 2007), pp. 93–102
114. M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W.H. Freeman, New York, 1979), 340 pp
115. V.K. Garg, *Elements of Distributed Computing* (Wiley-Interscience, New York, 2002), 423 pp
116. A.J. Gerber, Process synchronization by counter variables. ACM Operating Syst. Rev. **11**(4), 6–17 (1977)
117. A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, The NYU ultracomputer: designing an MIMD parallel computer. IEEE Trans. Comput. **C-32**(2), 175–189 (1984)
118. A. Gottlieb, B.D. Lubachevsky, L. Rudolph, Basic techniques for the efficient coordination of very large number of cooperating sequential processes. ACM Trans. Program. Lang. Syst. **5**(2), 164–189 (1983)
119. J. Gray, A. Reuter, *Transactions Processing: Concepts and Techniques* (Morgan Kaufmann, San Mateo, 1992), 1070 pp
120. R. Guerraoui, Indulgent algorithms. *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, Portland, 2000 (ACM Press, New York, 2000), pp. 289–298
121. R. Guerraoui, Th.A. Henzinger, V. Singh, Permissiveness in transactional memories. *Proceedings of the 22nd International Symposium on Distributed Computing (DISC'08)*, Arcachon, 2008. LNCS, vol. 5218 (Springer, Heidelberg, 2008), pp. 305–319

122. R. Guerraoui, M. Herlihy, B. Pochon, Towards a theory of transactional contention managers. *Proceedings of the 24th International ACM Symposium on Principles of Distributed Computing (PODC'05)*, Las Vegas, 2005 (ACM Press, New York, 2005), pp. 258–264
123. R. Guerraoui, M. Kapałka, On the correctness of transactional memory. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City, 2008 (ACM Press, New York, 2008), pp. 175–184
124. R. Guerraoui, M. Kapałka, Principles of transactional memory. In *Synthesis Lectures on Distributed Computing Theory* (Morgan and Claypool, San Rafael, 2010), 180 pp
125. R. Guerraoui, M. Kapalka, P. Kuznetsov, The weakest failure detectors to boost obstruction-freedom. Distrib. Comput. **20**(6), 415–433 (2008)
126. R. Guerraoui, N.A. Lynch, A general characterization of indulgence. ACM Trans. Auton. Adapt. Syst. **3**(4), Article 20, 2008
127. R. Guerraoui, M. Raynal, The information structure of indulgent consensus. IEEE Trans. Comput. **53**(4), 453–466 (2004)
128. R. Guerraoui, M. Raynal, A leader election protocol for eventually synchronous shared memory systems. *Proceedings of the 4th International IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'06)*, Seattle, 2006 (IEEE Computer Society Press, New York, 2006), pp. 75–80
129. R. Guerraoui, M. Raynal, The alpha of indulgent consensus. Comput. J. **50**(1), 53–67 (2007)
130. R. Guerraoui, M. Raynal, A universal construction for wait-free objects. *Proceedings of the ARES 2007 Workshop on Foundations of Fault-Tolerant Distributed Computing (FOFDC 2007)*, Vienna, 2007 (IEEE Society Computer Press, New York, 2007), pp. 959–966
131. R. Guerraoui, M. Raynal, From unreliable objects to reliable objects: the case of atomic registers and consensus. *Proceedings of the 9th International Conference on Parallel Computing Technologies (PaCT'07)*, Pereslavl-Zalessky, 2007. LNCS, vol. 4671 (Springer, Heidelberg, 2007), pp. 47–61
132. S. Haldar, P.S. Subramanian, Space-optimal conflict-free construction of 1-writer 1-reader multivalued atomic register. *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, Terschelling, 1994. LNCS, vol. 857 (Springer, Berlin, 1994), pp. 116–129
133. S. Haldar, K. Vidyasankar, Constructing 1-writer multireader multivalued atomic variables from regular variables. J. ACM **42**(1), 186–203 (1995)
134. T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, M. Valero, Transactional memory: an overview. IEEE Micro. **27**(3), 8–29 (2007)
135. T.L. Harris, K. Fraser, I.A. Pratt, A practical multi-word compare-and-swap operation. *Proceedings of the 16th Internationl Symposium on Distributed Computing (DISC'02)*, Toulouse, 2002. LNCS vol. 2508 (Springer, Heidelberg, 2002), pp. 265–279
136. J.-M. Hélary, A. Mostéfaoui, M. Raynal, Interval consistency of asynchronous distributed computations. J. Comput. Syst. Sci. **64**(2), 329–349 (2002)
137. S. Heller, M.P. Herlihy, V. Luchangco, M. Moir, W. Scherer III, N. Shavit, A lazy concurrent list-based algorithm. Parallel Process. Lett. **17**(4), 411–424 (2007)
138. M.P. Herlihy, Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
139. M.P. Herlihy, A methodology for implementing highly concurrent data objects. ACM Trans. Programm. Lang. Syst. **15**(5), 745–770 (1994)
140. M.P. Herlihy, V. Luchangco, Distributed computing and the multicore revolution. ACM SIGACT News **39**(1), 62–72 (2008)
141. M. Herlihy, V. Luchangco, M. Moir, W.M. Scherer III, Software transactional memory for dynamic-sized data structures. *Proceedings of the 22nd International ACM Symposium on Principles of Distributed Computing (PODC'03)*, Boston, 2003 (ACM Press, New York, 2003), pp. 92–101
142. M.P. Herlihy, V. Luchangco, P. Marin, M. Moir, Non-blocking memory management support for dynamic-size data structures. ACM Trans. Comput. Syst. **23**(2), 146–196 (2005)

143. M.P. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: double-ended queues as an example. *Proceedings of the 23rd International IEEE Conference on Distributed Computing Systems (ICDCS'03)*, Providence, 2003 (IEEE Press, New York, 2003), pp. 522–529
144. M.P. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures. *Proceedings of the 20th ACM International Symposium on Computer Architecture (ISCA'93)*, San Diego, 1993 (ACM Press, New York, 1993), pp. 289–300
145. M.P. Herlihy, N. Shavit, The topological structure of asynchronous computability. J. ACM **46**(6), 858–923 (1999)
146. M. Herlihy, N. Shavit, *The Art of Multiprocessor Programming* (Morgan Kaufmann, San Mateo, 2008), 508 pp. ISBN 978-0-12-370591-4
147. M. Herlihy, N. Shavit, On the nature of progress. *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS'11)*, Toulouse, 2011. LNCS, vol. 7109 (Springer, London, 2011), pp. 313–328
148. M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
149. C.E. Hewitt, R.R. Atkinson, Specification and proof techniques for serializers. IEEE Trans. Softw. Eng. **SE5**(1), 1–21 (1979)
150. C.A.R. Hoare, Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
151. J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 2nd edn. (Addison-Wesley, Reading, 2001), 521 pp
152. J.H. Howard, Proving monitors. Commun. ACM **19**(5), 273–279 (1976)
153. D. Imbs, J.R. de Mendivil, M. Raynal, Virtual world consistency: a new condition for STM systems. Brief announcement. *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*, Calgary, 2009 (ACM Press, New York, 2009), pp. 280–281
154. D. Imbs, S. Rajsbaum, M. Raynal, The universe of symmetry breaking tasks. *Proceedings of the 18th International Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, Gdansk, 2011. LNCS, vol. 6796 (Springer, Heidelberg, 2011), pp. 66–77
155. D. Imbs, M. Raynal, A lock-based STM protocol that satisfies opacity and progressiveness. *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, Luxor, 2008. LNCS, vol. 5401 (Springer, Heidelberg, 2008), pp. 226–245
156. D. Imbs, M. Raynal, Provable STM properties: leveraging clock and locks to favor commit and early abort. *10th International Conference on Distributed Computing and Networking (ICDCN'09)*, Hyderabad, January 2009. LNCS, vol. 5408 (Springer, Berlin, 2009), pp. 67–78
157. D. Imbs, M. Raynal, A note on atomicity: boosting test&set to solve consensus. Inf. Process. Lett. **109**(12), 589–591 (2009)
158. D. Imbs, M. Raynal, Help when needed, but no more: efficient read/write partial snapshot. J. Parallel Distrib. Comput. **72**(1), 1–13 (2012)
159. D. Imbs, M. Raynal, The multiplicative power of consensus numbers. *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, Zurich, 2010 (ACM Press, New York, 2010), pp. 26–35
160. D. Imbs, M. Raynal, On adaptive renaming under eventually limited contention. *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, New York, 2010. LNCS, vol. 6366 (Springer, New York, 2010), pp. 377–387
161. D. Imbs, M. Raynal, A liveness condition for concurrent objects: x-wait-freedom. Concurr. Comput. Pract. Exp. **23**, 2154–2166 (2011)
162. D. Imbs, M. Raynal, A simple snapshot algorithm for multicore systems. *Proceedings of the 5th IEEE Latin-American Symposium on Dependable Computing (LADC'11)*, 2011 (IEEE Press, New York, 2011), pp. 17–23
163. D. Imbs, M. Raynal, Virtual world consistency: a condition for STM systems (with a versatile protocol with invisible read operations). Theoret. Comput. Sci. **444**, 113–127 (2012)
164. D. Imbs, M. Raynal, G. Taubenfeld, On asymmetric progress conditions. *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, Zurich, 2010 (ACM Press, New York, 2010), pp. 55–64

165. D. Inoue, W. Chen, T. Masuzawa, N. Tokura, Linear time snapshot using multi-reader multi-writer registers. *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, Terschelling, 1994. LNCS, vol. 857 (Springer, London, 1994), pp. 130–140
166. P. Jayanti, Robust wait-free hierarchies. J. ACM **44**(4), 592–614 (1997)
167. P. Jayanti, An optimal multiwriter snapshot algorithm. *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC'05)*, Baltimore, 2005 (ACM Press, New York, 2005), pp. 723–732
168. P. Jayanti, J.E. Burns, G.L. Peterson, Almost optimal single reader single writer atomic register. J. Parallel Distrib. Comput. **60**, 150–168 (2000)
169. P. Jayanti, T.D. Chandra, S. Toueg, Fault-tolerant wait-free shared objects. J. ACM **45**(3), 451–500 (1998)
170. P. Jayanti, T.D. Chandra, S. Toueg, The cost of graceful degradation for omission failures. Inf. Process. Lett. **71**, 167–172 (1999)
171. P. Jayanti, K. Tan, G. Friedland, A. Katz, Bounding Lamport's bakery algorithm. *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'01)*, Piestany, 2004. LNCS, vol. 2234 (Springer, Berlin, 2004), pp. 261–270
172. P. Jayanti, S. Toueg, Some results on the impossibility, universality, and decidability of consensus. *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, Haifa, 1992. LNCS, vol. 647 (Springer, Heidelberg, 1992), pp. 69–84
173. N.D. Kallimanis, P. Fatourou, Revisiting the combining synchronization technique. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, New Orleans, 2012 (ACM Press, New York, 2012) pp. 257–266
174. J.L.W. Kessels, An alternative to event queues for synchronization in monitors. Commun. ACM **20**(7), 500–503 (1977)
175. J.L.W. Kessels, Arbitration without common modifiable variables. Acta Inform. **17**(2), 135–141 (1982)
176. J. Kleinberg, E. Tardos, *Algorithm Design* (Addison-Wesley, Pearson Education, New York, 2005), 838 pp
177. L.M. Kirousis, E. Kranakis, A survey of concurrent readers and writers. CWI Q. **2**, 307–330 (1989)
178. L.M. Kirousis, E. Kranakis, P. Vitányi, Atomic multireader register. *Proceedings of the 2nd International Workshop on Distributed Algorithms (WDAG'87)*, Amsterdam, 1987. LNCS, vol. 312 (Springer, Berlin, 1987), pp. 278–296
179. A. Kogan, E. Petrank, A methodology for creating fast wait-free data structures. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, New Orleans, 2012 (ACM Press, New York, 2012), pp. 141–150
180. C.P. Kruskal, L. Rudolph, M. Snir, Efficient synchronization on multiprocessors with shared memory. ACM Trans. Program. Lang. Syst. **10**(4), 579–601 (1988)
181. A. Kshemkalyani, M. Singhal, *Distributed Computing: Principles, Algorithms and Systems* (Cambridge University Press, Cambridge, 2008), 736 pp
182. E. Ladam-Mozes, N. Shavit, An optimistic approach to lock-free FIFO queues. *Proceedings of the 18th International Symposium on Distributed Computing (DISC'04)*, Amsterdam, 2004. LNCS, vol. 3274 (Springer, Heidelberg, 2004), pp. 117–131
183. L. Lamport, A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
184. L. Lamport, Concurrent reading while writing. Commun. ACM **20**(11), 806–811 (1977)
185. L. Lamport, Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **SE-3**(2), 125–143 (1977)
186. L. Lamport, Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
187. L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **C28**(9), 690–691 (1979)

188. L. Lamport, The mutual exclusion problem. Part I: a theory of interprocess communication, Part II: statement and solutions. J. ACM **33**, 313–348 (1986)
189. L. Lamport, On interprocess communication, Part I: basic formalism. Distrib. Comput. **1**(2), 77–85 (1986)
190. L. Lamport, On interprocess communication, Part II: algorithms. Distrib. Comput. **1**(2), 77–101 (1986)
191. L. Lamport, Fast mutual exclusion. ACM Trans. Comput. Syst. **5**(1), 1–11 (1987)
192. L. Lamport, The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998) (first version appeared as DEC Research, Report #49, September 1989)
193. L. Lamport, Arbitration-free synchronization. Distrib. Comput. **16**(2–3), 219–237 (2003)
194. L. Lamport, Teaching concurrency. ACM SIGACT News Distrib. Comput. Column **40**(1), 58–62 (2009)
195. J. Larus, Ch. Kozyrakis, Transactional memory: is TM the answer for improving parallel programming? Commun. ACM **51**(7), 80–89 (2008)
196. M. Li, J. Tromp, P. Vitányi, How to share concurrent wait-free variables. J. ACM **43**(4), 723–746 (1996)
197. W.-K. Lo, V. Hadzilacos, Using failure detectors to solve consensus in asynchronous shared memory systems. *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, 1994. LNCS, vol. 857 (Springer, Heidelberg, 1994), pp. 280–295
198. W.-K. Lo, V. Hadzilacos, All of us are smarter than any of us: wait-free hierarchies are not robust. *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC'97)*, El Paso, 1997 (ACM Press, New York, 1997), pp. 579–588
199. M. Loui, H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processes. Adv. Comput. Res. **4**, 163–183 (1987). JAI Press
200. V. Luchangco, D. Nussbaum, N. Shavit, A hierarchical CLH queue lock. *Proceedings of the 12th European Conference on Parallel Computing (Euro-Par'06)*, Dresden, 2006. LNCS, vol. 4128 (Springer, Berlin, 2006), pp. 801–810
201. N.A. Lynch, *Distributed Algorithms* (Morgan Kaufmann, San Mateo, 1996), 872 pp
202. F. Mattern, Virtual time and global states in distributed computations. ed. by M. Cosnard, P. Quinton, M. Raynal, Y. Robert, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, North-Holland, 1989, pp. 215–226
203. K. Mehlhorn, P. Sanders, *Algorithms and Data Structures* (Springer, Berlin, 2008), 300 pp
204. M. Merritt, G. Taubenfeld, Computing with infinitely many processes. *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, Toledo, 2000. LNCS, vol. 1914 (Springer, Heidelberg, 2000), pp. 164–178
205. M.M. Michael, M.L. Scott, Simple, fast and practical blocking and non-blocking concurrent queue algorithms. *Proceedings of the 15th International ACM Symposium on Principles of Distributed Computing (PODC'96)*, Philadelphia, 1996 (ACM Press, New York, 1996), pp. 267–275
206. J. Misra, Axioms for memory access in asynchronous hardware systems. ACM Trans. Program. Lang. Syst. **8**(1), 142–153 (1986)
207. M. Moir, Practical implementation of non-blocking synchronization primitives. *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, Santa Barbara, 1997 (ACM Press, New York, 1997), pp. 219–228
208. M. Moir, Fast, long-lived renaming improved and simplified. Sci. Comput. Program. **30**, 287–308 (1998)
209. M. Moir, J. Anderson, Wait-free algorithms for fast, long-lived renaming. Sci. Comput. Program. **25**(1), 1–39 (1995)
210. M. Moir, D. Nussbaum, O. Shalev, N. Shavit, Using elimination to implement scalable and lock-free FIFO queues. *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'05)*, Las Vegas, 2005 (ACM Press, New York, 2005), pp. 253–262
211. B. Moret, *The Theory of Computation* (Addison-Wesley, Reading, 1998), 453 pp
212. A. Mostéfaoui, M. Raynal, Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, Bratislava, 1999. LNCS, vol. 1693 (Springer, Berlin, 1999), 4963 pp

213. A. Mostéfaoui, M. Raynal, Leader-based consensus. Parallel Process. Lett. **11**(1), 95–107 (2001)
214. A. Mostéfaoui, M. Raynal, Looking for efficient implementations of concurrent objects. *Proceedings of the 11th International Conference on Parallel Computing Technologies (PaCT'11)*, Kazan, 2011. LNCS, vol. 6873 (Springer, Berlin, 2011), pp. 74–87
215. M. Mostéfaoui, M. Raynal, C. Travers, Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement. *Proceedings of the 20th International Symposium on Distributed Computing (DISC'09)*, Elche, 2009. LNCS, vol. 4167 (Springer, Heidelberg, 2006), pp. 1–15
216. A. Mostéfaoui, M. Raynal, C. Travers, From renaming to k-set agreement. *14th International Colloquium on Structural Information and Communication Complexity (SIROCCO'07)*, Castiglioncello, 2007, LNCS, vol. 4474 (Springer, Berlin, 2007), pp. 62–76
217. A. Mostéfaoui, M. Raynal, F. Tronel, From binary consensus to multivalued consensus in asynchronous message-passing systems. Inf. Process. Lett. **73**, 207–213 (2000)
218. G. Neiger, Failure detectors and the wait-free hierarchy. *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, Ottawa, 1995 (ACM Press, New York, 1995), pp. 100–109
219. R. Newman-Wolfe, A protocol for wait-free atomic multi-reader shared variables. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC'87)*, Vancouver, 1987 (ACM Press, New York, 1987), pp. 232–248
220. S. Owicki, D. Gries, Verifying properties of parallel programs. Commun. ACM **19**(5), 279–285 (1976)
221. Ch. Papadimitriou, The serializability of concurrent updates. J. ACM **26**(4), 631–653 (1979)
222. Ch. Papadimitriou, *The Theory of Database Concurrency Control* (Computer Science Press, Cambridge, 1988), 239 pp
223. D. Perelman, R. Fan, I. Keidar, On maintaining multiple versions in STM. *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC'10)*, Zurich, 2010 (ACM Press, New York, 2010), pp. 16–25
224. G.L. Peterson, Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
225. G.L. Peterson, Concurrent reading while writing. ACM Trans. Program. Lang. Syst. **5**, 46–55 (1983)
226. G.L. Peterson, R. Bazzi, N. Neiger, A gap theorem for consensus types (extended abstract). *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, Los Angeles, 1994 (ACM Press, New York, 1994), pp. 344–353
227. G.L. Peterson, M.J. Fisher, Economical solutions for the critical section problem in distributed systems. *Proceedings of the 9th ACM Symposium on Theory of Computing (STOC'77)*, Boulder, 1977 (ACM Press, New York, 1977), pp. 91–97
228. S.A. Plotkin, Sticky bits and universality of consensus. *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, Edmonton, 1989 (ACM Press, New York, 1989), pp. 159–176
229. S. Rajsbaum, M. Raynal, A theory-oriented introduction to wait-free synchronization based on the adaptive renaming problem. *Proceedings of the 25th International Conference on Advanced Information Networking and Applications (AINA'11)*, Singapore, 2011 (IEEE Press, New York, 2011), pp. 356–363
230. S. Rajsbaum, M. Raynal, C. Travers, The iterated restricted immediate snapshot model. *Proceedings of the 14th Annual International Conference on Computing and Combinatorics (COCOON'08)*, Dalian, 2008. LNCS, vol. 5092 (Springer, Heidelberg, 2008), pp. 487–497
231. M. Raynal, *Algorithms for Mutual Exclusion* (The MIT Press, New York, 1986), 107 pp. ISBN 0-262-18119-3
232. M. Raynal, Sequential consistency as lazy linearizability. *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, Winnipeg, 2002 (ACM Press, New York, 2002), pp. 151–152
233. M. Raynal, Token-based sequential consistency. Int. J. Comput. Syst. Sci. Eng. **17**(6), 359–366 (2002)

234. M. Raynal, Reliable compare&swap for fault-tolerant synchronization. *Proceedings of the 8th International IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, Guadalajara, 2003 (IEEE Computer Society Press, New York, 2003), pp. 50–55
235. M. Raynal, Failure detectors for asynchronous distributed systems: an introduction. Wiley Encycl. Comput. Sci. Eng. **2**, 1181–1191 (2009)
236. M. Raynal, *Communication and Agreement Abstractions for Fault-tolerant Asynchronous Distributed Systems* (Morgan and Claypool, San Rafael, 2010), 251 pp. ISBN 9781608452934
237. M. Raynal, *Fault-tolerant Agreement in Synchronous Distributed Systems* (Morgan and Claypool, San Rafael, 2010), 167 pp. ISBN 9781608455256
238. M. Raynal, On the implementation of concurrent objects. *Dependable and Historic Computing (Randell's Tales: a Festschrift recognizing the contributions of Brian Randell)*, LNCS, vol. 6875 (Springer, Berlin, 2011), pp. 453–478
239. M. Raynal, J. Stainer, From the Ω and store-collect building blocks to efficient asynchronous consensus. *Proceeedings of the 18th International European Parallel Computing Conference (Euro-Par'12)*, 2012. LNCS, vol. 7484 (Springer, Berlin, 2012), pp. 427–438
240. M. Raynal, J. Stainer, A simple asynchronous shared memory consensus algorithm based on Ω and closing sets. *Proceedings of the 6th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS'12)*, Palermo, 2012 (IEEE Press, New York, 2012), pp. 357–364
241. M. Raynal, G. Thia-Kime, M. Ahamad, From serializable to causal transactions. *Brief announcement, Proceedings of the 15th ACM Symposium on Distributed Computing (PODC'96)*, Philadelphia, 1996 (ACM Press, New York, 1996), 310 pp
242. M. Raynal, C. Travers, In search of the Holy Grail: looking for the weakest failure detector for wait-free set agreement. *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS'11)*, Toulouse, 2006. LNCS, vol. 4305 (Springer, Heidelberg, 2006), pp. 3–19
243. D.P. Reed, R.K. Kanodia, Synchronization with eventcounts and sequencers. Commun. ACM **22**(2), 115–123 (1979)
244. T. Riegel, P. Felber, C. Fetzer, A lazy snapshot algorithm with eager validation. *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, Stockholm, 2006. LNCS, vol. 4167 (Springer, Heidelberg, 2006), pp. 284–298
245. T. Riegel, C. Fetzer, P. Felber, Time-based transactional memory with scalable time bases. *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, San Diego, 2007 (ACM Press, New York, 2007), pp. 221–228
246. P. Robert, J.-P. Verjus, Towards autonomous description of synchronization modules. *Proceedings of the IFIP World Congress*, Toronto, 1977, pp. 981–986
247. L.R. Sampaio, F.V. Brasileiro, Adaptive indulgent consensus. *Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN 2005)*, 2005 (IEEE Computer Society Press, New York, 2005), pp. 422–431
248. N. Santoro, *Design and Analysis of Distributed Algorithms* (Wiley, New York, 2007), 589 pp
249. W.N. Scherer III, M.L. Scott, Advanced contention management in dynamic software transactional memory. *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, Las Vegas, 2005 (ACM Press, New York, 2005), pp. 240–248
250. F.B. Schneider, Implementing fault-tolerant services using the state machine approach. ACM Comput. Surv. **22**(4), 299–319 (1990)
251. R. Schwarz, F. Mattern, Detecting causal relationship in distributed computations: in search of the Holy Grail. Distrib. Comput. **7**, 149–174 (1993)
252. L.M. Scott, Sequential specification of transactional memory semantics. *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, Ottawa, 2006 (ACM Press, New York, 2006)
253. N. Shafiei, Non-blocking array-based algorithms for stacks and queues. *Proceedings of the 10th International Conference on Distributed Computing and Networking (ICDCN'09)*, Hyderabad, 2009. LNCS, vol. 5408 (Springer, Heidelberg, 2009), pp. 55–66

254. C. Shao, E. Pierce, J. Welch, Multi-writer consistency conditions for shared memory objects. *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, Sorrento, 2003. LNCS, vol. 2848 (Springer, Heidelberg, 2003), pp. 106–120
255. N. Shavit, D. Touitou, Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
256. E. Shenk, The consensus hierarchy is not robust. *Proceedings 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, Santa Barbara, 1997 (ACM Press, New York, 1997), p. 279
257. A.K. Singh, J.H. Anderson, M.G. Gouda, The elusive atomic register. J. ACM **41**(2), 311–339 (1994)
258. M. Sipser, *Introduction to the Theory of Computation* (PWS, Boston, 1996), 396 pp
259. M.F. Spear, V.J. Marathe, W.N. Scherer III, M.L. Scott, Conflict detection and validation strategies for software transactional memory. *Proceedings of the 20th Symposium on Distributed Computing (DISC'06)*, Stockholm, 2006. LNCS, vol. 4167 (Springer, Heidelberg, 2006), pp. 179–193
260. H.S. Stone, Database applications of the fetch&add instruction. IEEE Trans. Comput. **C-33**(7), 604–612 (1984)
261. G. Taubenfeld, The black-white bakery algorithm. *Proceedings of the 18th International Symposium on Distributed Computing (DISC'04)*, Amsterdam, 2004. LNCS, vol. 3274 (Springer, Heidelberg, 2004), pp. 56–70
262. G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming* (Pearson Education/Prentice Hall, Upper Saddle River, 2006), 423 pp. ISBN 0-131-97259-6
263. G. Taubenfeld, Contention-sensitive data structure and algorithms. *Proceedings of the 23rd International Symposium on Distributed Computing (DISC'09)*, Elche, 2009. LNCS, vol. 5805 (Springer, Heidelberg, 2009), pp. 157–171
264. G. Taubenfeld, The computational structure of progress conditions. *Proceedings of the 24th International Symposium on Distributed Computing (DISC'10)*, Cambridge, 2010. LNCS, vol. 6343 (Springer, Heidelberg, 2010), pp. 221–235
265. J. Tromp, How to construct an atomic variable. *Proceedings of the 3rd International Workshop on Distributed Algorithms (WDAG'89)*, Nice, 1989. LNCS, vol. 392 (Springer, Heidelberg, 1989), pp. 292–302
266. Ph. Tsigas, Y. Zhang, A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. *Proceedings of the 13th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'01)*, Heraklion, 2001 (ACM Press, New York), pp. 134–143
267. J.D. Valois, Implementing lock-free queues. *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems (PDCS'94)*, Las Vegas, 1994 (IEEE Press, New York, 1994), pp. 64–69
268. K. Vidyasankar, Converting Lamport's regular register to atomic register. Inf. Proces. Lett. **30**, 221–223 (1989)
269. K. Vidyasankar, An elegant 1-writer multireader multivalued atomic register. Inf. Proces. Lett. **30**(5), 221–223 (1989)
270. K. Vidyasankar, Concurrent reading while writing revisited. Distrib. Comput. **4**, 81–85 (1990)
271. K. Vidyasankar, A very simple construction of 1-writer multireader multivalued atomic variable. Inf. Proces. Lett. **37**, 323–326 (1991)
272. P. Vitányi, B. Awerbuch, Atomic shared register access by asynchronous hardware. *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science (FOCS'87)*, Los Angeles, 1987 (IEEE Press, New York, 1987), pp. 223–243, 1987 (errata, ibid, 1987)
273. J.-T. Wamhoff, Ch. Fetzer, The universal transactional memory construction. Technical Report, 12 pp, University of Dresden (Germany), 2010
274. W. Wu, J. Cao, J. Yang, M. Raynal, Using asynchrony and zero degradation to speed up indulgent consensus protocols. J. Parallel Distrib. Comput. **68**(7), 984–996 (2008)
275. J. Yang, G. Neiger, E. Gafni, Structured derivations of consensus algorithms for failure detectors. *Proceedings of the 17th Symposium on Principles of Distributed Computing (PODC)*, Puerto Vallarta, 1998 (ACM Press, New York, 1998), pp. 297–308

Index