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 (–$\LaTeX 2_\epsilon$ 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

Part II On the Foundations Side: The Atomicity Concept

Part III Mutex-Free Synchronization

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 $i \in \{1, \ldots, m\}$ **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).

M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*,
DOI: 10.1007/978-3-642-32027-9_1,

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 ← 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

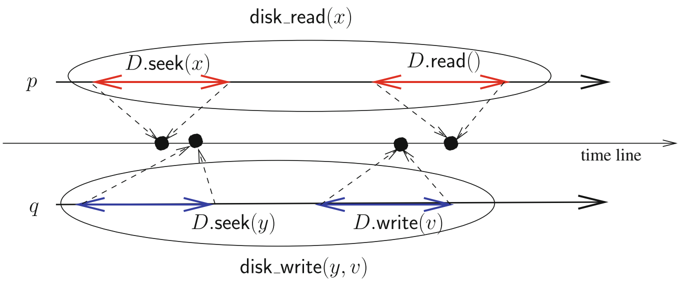

Fig. 1.2 An interleaving of invocations to disk primitives

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

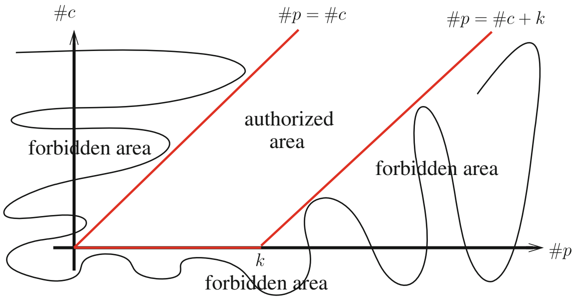

Fig. 1.3 Synchronization is to preserve invariants

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.

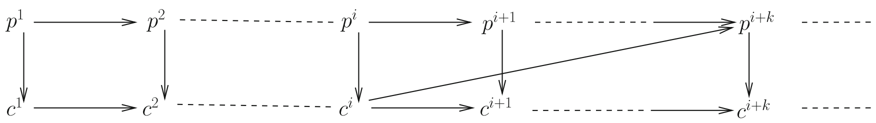

Fig. 1.4 Invariant expressed with control flows

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 $\big(LOCK.$acquire_lock(); $LOCK.$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.

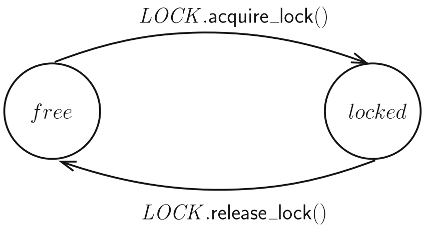

Fig. 1.5 Sequential specification of a lock object *LOCK*

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].