

Concurrent Programming: Algorithms, Principles, and Foundations

Michel Raynal

Concurrent Programming: Algorithms, Principles, and Foundations

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

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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), Ajay 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 (– $\text{\LaTeX}2_{\epsilon}$ 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 I Lock-Based Synchronization

1	The Mutual Exclusion Problem.	3
1.1	Multiprocess Program	3
1.1.1	The Concept of a Sequential Process.	3
1.1.2	The Concept of a Multiprocess Program	4
1.2	Process Synchronization	4
1.2.1	Processors and Processes	4
1.2.2	Synchronization	4
1.2.3	Synchronization: Competition.	5
1.2.4	Synchronization: Cooperation.	7
1.2.5	The Aim of Synchronization	
	Is to Preserve Invariants	7
1.3	The Mutual Exclusion Problem	9
1.3.1	The Mutual Exclusion Problem (Mutex)	9
1.3.2	Lock Object.	11
1.3.3	Three Families of Solutions	12
1.4	Summary	13
1.5	Bibliographic Notes	13
2	Solving Mutual Exclusion	15
2.1	Mutex Based on Atomic Read/Write Registers.	15
2.1.1	Atomic Register	15
2.1.2	Mutex for Two Processes:	
	An Incremental Construction	17
2.1.3	A Two-Process Algorithm	19
2.1.4	Mutex for n Processes:	
	Generalizing the Previous Two-Process Algorithm. . .	22
2.1.5	Mutex for n Processes:	
	A Tournament-Based Algorithm.	26
2.1.6	A Concurrency-Abortable Algorithm.	29

2.1.7	A Fast Mutex Algorithm	33
2.1.8	Mutual Exclusion in a Synchronous System.	37
2.2	Mutex Based on Specialized Hardware Primitives	38
2.2.1	Test&Set, Swap and Compare&Swap	39
2.2.2	From Deadlock-Freedom to Starvation-Freedom.	40
2.2.3	Fetch&Add	44
2.3	Mutex Without Atomicity	45
2.3.1	Safe, Regular and Atomic Registers	45
2.3.2	The Bakery Mutex Algorithm	48
2.3.3	A Bounded Mutex Algorithm.	53
2.4	Summary	58
2.5	Bibliographic Notes	58
2.6	Exercises and Problems	59
3	Lock-Based Concurrent Objects	61
3.1	Concurrent Objects	61
3.1.1	Concurrent Object.	61
3.1.2	Lock-Based Implementation.	62
3.2	A Base Synchronization Object: the Semaphore	63
3.2.1	The Concept of a Semaphore	63
3.2.2	Using Semaphores to Solve the Producer-Consumer Problem.	65
3.2.3	Using Semaphores to Solve a Priority Scheduling Problem	71
3.2.4	Using Semaphores to Solve the Readers-Writers Problem	74
3.2.5	Using a Buffer to Reduce Delays for Readers and Writers.	78
3.3	A Construct for Imperative Languages: the Monitor	81
3.3.1	The Concept of a Monitor	82
3.3.2	A Rendezvous Monitor Object	83
3.3.3	Monitors and Predicates.	85
3.3.4	Implementing a Monitor from Semaphores	87
3.3.5	Monitors for the Readers-Writers Problem.	89
3.3.6	Scheduled Wait Operation	94
3.4	Declarative Synchronization: Path Expressions.	95
3.4.1	Definition	96
3.4.2	Using Path Expressions to Solve Synchronization Problems	97
3.4.3	A Semaphore-Based Implementation of Path Expressions.	98
3.5	Summary	101
3.6	Bibliographic Notes	102
3.7	Exercises and Problems	102

Part II On the Foundations Side: The Atomicity Concept

4	Atomicity: Formal Definition and Properties	113
4.1	Introduction	113
4.2	Computation Model	115
4.2.1	Processes and Operations	115
4.2.2	Objects	116
4.2.3	Histories	117
4.2.4	Sequential History	119
4.3	Atomicity	120
4.3.1	Legal History	120
4.3.2	The Case of Complete Histories	121
4.3.3	The Case of Partial Histories	123
4.4	Object Composability and Guaranteed Termination Property	125
4.4.1	Atomic Objects Compose for Free	125
4.4.2	Guaranteed Termination	127
4.5	Alternatives to Atomicity	128
4.5.1	Sequential Consistency	128
4.5.2	Serializability	130
4.6	Summary	131
4.7	Bibliographic Notes	132

Part III Mutex-Free Synchronization

5	Mutex-Free Concurrent Objects	135
5.1	Mutex-Freedom and Progress Conditions	135
5.1.1	The Mutex-Freedom Notion	135
5.1.2	Progress Conditions	137
5.1.3	Non-blocking with Respect to Wait-Freedom	140
5.2	Mutex-Free Concurrent Objects	140
5.2.1	The Splitter: A Simple Wait-Free Object from Read/Write Registers	140
5.2.2	A Simple Obstruction-Free Object from Read/Write Registers	143
5.2.3	A Remark on Compare&Swap: The ABA Problem	145
5.2.4	A Non-blocking Queue Based on Read/Write Registers and Compare&Swap	146
5.2.5	A Non-blocking Stack Based on Compare&Swap Registers	150
5.2.6	A Wait-Free Stack Based on Fetch&Add and Swap Registers	152

5.3	Boosting Obstruction-Freedom to Stronger Progress in the Read/Write Model	155
5.3.1	Failure Detectors	155
5.3.2	Contention Managers for Obstruction-Free Object Implementations	157
5.3.3	Boosting Obstruction-Freedom to Non-blocking	158
5.3.4	Boosting Obstruction-Freedom to Wait-Freedom	159
5.3.5	Mutex-Freedom Versus Loops Inside a Contention Manager Operation	161
5.4	Summary	162
5.5	Bibliographic Notes	162
5.6	Exercises and Problems	163
6	Hybrid Concurrent Objects	165
6.1	The Notion of a Hybrid Implementation	165
6.1.1	Lock-Based Versus Mutex-Free Operation: Static Hybrid Implementation.	166
6.1.2	Contention Sensitive (or Dynamic Hybrid) Implementation.	166
6.1.3	The Case of Process Crashes	166
6.2	A Static Hybrid Implementation of a Concurrent Set Object	167
6.2.1	Definition and Assumptions	167
6.2.2	Internal Representation and Operation Implementation.	167
6.2.3	Properties of the Implementation	171
6.3	Contention-Sensitive Implementations	172
6.3.1	Contention-Sensitive Binary Consensus	172
6.3.2	A Contention Sensitive Non-blocking Double-Ended Queue	176
6.4	The Notion of an Abortable Object.	181
6.4.1	Concurrency-Abortable Object	181
6.4.2	From a Non-blocking Abortable Object to a Starvation-Free Object	183
6.5	Summary	186
6.6	Bibliographic Notes	186
6.7	Exercises and Problems	187
7	Wait-Free Objects from Read/Write Registers Only	189
7.1	A Wait-Free Weak Counter for Infinitely Many Processes.	189
7.1.1	A Simple Counter Object.	190
7.1.2	Weak Counter Object for Infinitely Many Processes.	191
7.1.3	A One-Shot Weak Counter Wait-Free Algorithm	193
7.1.4	Proof of the One-Shot Implementation	194
7.1.5	A Multi-Shot Weak Counter Wait-Free Algorithm	199

7.2	Store-Collect Object	201
7.2.1	Store-Collect Object: Definition	201
7.2.2	An Adaptive Store-Collect Implementation	204
7.2.3	Proof and Cost of the Adaptive Implementation	208
7.3	Fast Store-Collect Object	211
7.3.1	Fast Store-Collect Object: Definition.	211
7.3.2	A Fast Algorithm for the <code>store_collect()</code> Operation.	212
7.3.3	Proof of the Fast Store-Collect Algorithm	215
7.4	Summary	217
7.5	Bibliographic Notes	217
7.6	Problem.	218
8	Snapshot Objects from Read/Write Registers Only	219
8.1	Snapshot Objects: Definition	219
8.2	Single-Writer Snapshot Object	220
8.2.1	An Obstruction-Free Implementation.	221
8.2.2	From Obstruction-Freedom to Bounded Wait-Freedom	223
8.2.3	One-Shot Single-Writer Snapshot Object: Containment Property	227
8.3	Single-Writer Snapshot Object with Infinitely Many Processes	228
8.4	Multi-Writer Snapshot Object.	230
8.4.1	The Strong Freshness Property	231
8.4.2	An Implementation of a Multi-Writer Snapshot Object	231
8.4.3	Proof of the Implementation.	234
8.5	Immediate Snapshot Objects	238
8.5.1	One-Shot Immediate Snapshot Object: Definition	238
8.5.2	One-Shot Immediate Snapshot Versus One-Shot Snapshot	238
8.5.3	An Implementation of One-Shot Immediate Snapshot Objects	240
8.5.4	A Recursive Implementation of a One-Shot Immediate Snapshot Object	244
8.6	Summary	247
8.7	Bibliographic Notes	247
8.8	Problem.	248

9	Renaming Objects from Read/Write Registers Only	249
9.1	Renaming Objects.	249
9.1.1	The Base Renaming Problem	249
9.1.2	One-Shot Renaming Object	250
9.1.3	Adaptive Implementations	250
9.1.4	A Fundamental Result	251
9.1.5	Long-Lived Renaming.	252
9.2	Non-triviality of the Renaming Problem	252
9.3	A Splitter-Based Optimal Time-Adaptive Implementation	254
9.4	A Snapshot-Based Optimal Size-Adaptive Implementation.	256
9.4.1	A Snapshot-Based Implementation	256
9.4.2	Proof of the Implementation.	258
9.5	Recursive Store-Collect-Based Size-Adaptive Implementation.	259
9.5.1	A Recursive Renaming Algorithm	259
9.5.2	An Example.	262
9.5.3	Proof of the Renaming Implementation	263
9.6	Variant of the Previous Recursion-Based Renaming Algorithm.	266
9.6.1	A Renaming Implementation Based on Immediate Snapshot Objects	266
9.6.2	An Example of a Renaming Execution	268
9.7	Long-Lived Perfect Renaming Based on Test&Set Registers.	269
9.7.1	Perfect Adaptive Renaming	269
9.7.2	Perfect Long-Lived Test&Set-Based Renaming	270
9.8	Summary	271
9.9	Bibliographic Notes	271
9.10	Exercises and Problems	272

Part IV The Transactional Memory Approach

10	Transactional Memory	277
10.1	What Are Software Transactional Memories	277
10.1.1	Transactions = High-Level Synchronization	277
10.1.2	At the Programming Level.	279
10.2	STM System	281
10.2.1	Speculative Executions, Commit and Abort of a Transaction	281
10.2.2	An STM Consistency Condition: Opacity	282
10.2.3	An STM Interface.	282
10.2.4	Incremental Reads and Deferred Updates.	283

	10.2.5	Read-Only Versus Update Transactions	283
	10.2.6	Read Invisibility	284
10.3		A Logical Clock-Based STM System: TL2	284
	10.3.1	Underlying System and Control Variables of the STM System	284
	10.3.2	Underlying Principle: Consistency with Respect to Transaction Birth Date	285
	10.3.3	The Implementation of an Update Transaction	286
	10.3.4	The Implementation of a Read-Only Transaction	288
10.4		A Version-Based STM System: JVSTM	289
	10.4.1	Underlying and Control Variables of the STM System	290
	10.4.2	The Implementation of an Update Transaction	291
	10.4.3	The Implementation of a Read-Only Transaction	293
10.5		A Vector Clock-Based STM System	293
	10.5.1	The Virtual World Consistency Condition	293
	10.5.2	An STM System for Virtual World Consistency	295
	10.5.3	The Algorithms Implementing the STM Operations	296
10.6		Summary	299
10.7		Bibliographic Notes	299
10.8		Exercises and Problems	300

Part V On the Foundations Side: From Safe Bits to Atomic Registers

11		Safe, Regular, and Atomic Read/Write Registers	305
11.1		Safe, Regular, and Atomic Registers	305
	11.1.1	Reminder: The Many Faces of a Register	305
	11.1.2	From Regularity to Atomicity: A Theorem	308
	11.1.3	A Fundamental Problem: The Construction of Registers	310
11.2		Two Very Simple Bounded Constructions	311
	11.2.1	Safe/Regular Registers: From Single-Reader to Multi-Reader	311
	11.2.2	Binary Multi-Reader Registers: From Safe to Regular	313
11.3		From Bits to b -Valued Registers	314
	11.3.1	From Safe Bits to b -Valued Safe Registers	314
	11.3.2	From Regular Bits to Regular b -Valued Registers	315
	11.3.3	From Atomic Bits to Atomic b -Valued Registers	319

11.4	Three Unbounded Constructions	321
11.4.1	SWSR Registers: From Unbounded Regular to Atomic.	322
11.4.2	Atomic Registers: From Unbounded SWSR to SWMR	324
11.4.3	Atomic Registers: From Unbounded SWMR to MWMR	325
11.5	Summary	327
11.6	Bibliographic Notes	327
12	From Safe Bits to Atomic Bits:	
	Lower Bound and Optimal Construction	329
12.1	A Lower Bound Theorem	329
12.1.1	Two Preliminary Lemmas	330
12.1.2	The Lower Bound Theorem	331
12.2	A Construction of an Atomic Bit from Three Safe Bits	334
12.2.1	Base Architecture of the Construction	334
12.2.2	Underlying Principle and Signaling Scheme	335
12.2.3	The Algorithm Implementing the Operation $R.write()$	336
12.2.4	The Algorithm Implementing the Operation $R.read()$	336
12.2.5	Cost of the Construction	338
12.3	Proof of the Construction of an Atomic Bit	338
12.3.1	A Preliminary Theorem	338
12.3.2	Proof of the Construction	340
12.4	Summary	344
12.5	Bibliographic Notes	345
12.6	Exercise	345
13	Bounded Constructions of Atomic b-Valued Registers	347
13.1	Introduction	347
13.2	A Collision-Free (Pure Buffers) Construction	349
13.2.1	Internal Representation of the Atomic b -Valued Register R	349
13.2.2	Underlying Principle: Two-Level Switch to Ensure Collision-Free Accesses to Buffers	349
13.2.3	The Algorithms Implementing the Operations $R.write()$ and $R.read()$	350
13.2.4	Proof of the Construction: Collision-Freedom	352
13.2.5	Correctness Proof	355
13.3	A Construction Based on Impure Buffers	357
13.3.1	Internal Representation of the Atomic b -Valued Register R	357

13.3.2	An Incremental Construction	358
13.3.3	The Algorithms Implementing the Operations $R.write()$ and $R.read()$	360
13.3.4	Proof of the Construction.	360
13.3.5	From SWSR to SWMR b -Valued Atomic Register	367
13.4	Summary	368
13.5	Bibliographic Notes	368

Part VI On the Foundations Side:

The Computability Power of Concurrent Objects (Consensus)

14	Universality of Consensus	371
14.1	Universal Object, Universal Construction, and Consensus Object	371
14.1.1	Universal (Synchronization) Object and Universal Construction	371
14.1.2	The Notion of a Consensus Object	372
14.2	Inputs and Base Principles of Universal Constructions	373
14.2.1	The Specification of the Constructed Object	373
14.2.2	Base Principles of Universal Constructions	374
14.3	An Unbounded Wait-Free Universal Construction.	374
14.3.1	Principles and Description of the Construction	375
14.3.2	Proof of the Construction.	378
14.3.3	Non-deterministic Objects	382
14.3.4	Wait-Freedom Versus Bounded Wait-Freedom.	383
14.4	A Bounded Wait-Free Universal Construction	384
14.4.1	Principles of the Construction	384
14.4.2	Proof of the Construction.	388
14.4.3	Non-deterministic Objects	391
14.5	From Binary Consensus to Multi-Valued Consensus	391
14.5.1	A Construction Based on the Bit Representation of Proposed Values.	392
14.5.2	A Construction for Unbounded Proposed Values	394
14.6	Summary	395
14.7	Bibliographic Notes	396
14.8	Exercises and Problems	396
15	The Case of Unreliable Base Objects.	399
15.1	Responsive Versus Non-responsive Crash Failures	400
15.2	SWSR Registers Prone to Crash Failures.	400
15.2.1	Reliable Register When Crash Failures Are Responsive: An Unbounded Construction	401

15.2.2	Reliable Register When Crash Failures Are Responsive: A Bounded Construction	403
15.2.3	Reliable Register When Crash Failures Are Not Responsive: An Unbounded Construction	406
15.3	Consensus When Crash Failures Are Responsive: A Bounded Construction	408
15.3.1	The “Parallel Invocation” Approach Does Not Work	408
15.3.2	A t -Tolerant Wait-Free Construction	409
15.3.3	Consensus When Crash Failures Are Not Responsive: An Impossibility	410
15.4	Omission and Arbitrary Failures	410
15.4.1	Object Failure Modes	410
15.4.2	Simple Examples	412
15.4.3	Graceful Degradation	413
15.4.4	Fault-Tolerance Versus Graceful Degradation	417
15.5	Summary	418
15.6	Bibliographic Notes	419
15.7	Exercises and Problems	419
16	Consensus Numbers and the Consensus Hierarchy	421
16.1	The Consensus Number Notion	421
16.2	Fundamentals	422
16.2.1	Schedule, Configuration, and Valence	422
16.2.2	Bivalent Initial Configuration	423
16.3	The Weak Wait-Free Power of Atomic Registers	425
16.3.1	The Consensus Number of Atomic Read/Write Registers Is 1	425
16.3.2	The Wait-Free Limit of Atomic Registers	428
16.4	Objects Whose Consensus Number Is 2.	429
16.4.1	Consensus from Test&Set Objects	429
16.4.2	Consensus from Queue Objects	431
16.4.3	Consensus from Swap Objects	432
16.4.4	Other Objects for Wait-Free Consensus in a System of Two Processes	432
16.4.5	Power and Limit of the Previous Objects.	433
16.5	Objects Whose Consensus Number Is $+\infty$	438
16.5.1	Consensus from Compare&Swap Objects	439
16.5.2	Consensus from Mem-to-Mem-Swap Objects	440
16.5.3	Consensus from an Augmented Queue	442
16.5.4	From a Sticky Bit to Binary Consensus	442
16.5.5	Impossibility Result	443

16.6	Hierarchy of Atomic Objects	443
16.6.1	From Consensus Numbers to a Hierarchy	443
16.6.2	On Fault Masking	444
16.6.3	Robustness of the Hierarchy	445
16.7	Summary	445
16.8	Bibliographic Notes	445
16.9	Exercises and Problems	446
17	The Alpha(s) and Omega of Consensus:	
	Failure Detector-Based Consensus	449
17.1	De-constructing Compare&Swap	450
17.2	A Liveness-Oriented Abstraction: The Failure Detector Ω	452
17.2.1	Definition of Ω	452
17.2.2	Ω -Based Consensus: Ω as a Resource Allocator or a Scheduler	453
17.3	Three Safety-Oriented Abstractions: Alpha ₁ , Alpha ₂ , and Alpha ₃	454
17.3.1	A Round-Free Abstraction: Alpha ₁	454
17.3.2	A Round-Based Abstraction: Alpha ₂	455
17.3.3	Another Round-Free Abstraction: Alpha ₃	456
17.3.4	The Rounds Seen as a Resource	457
17.4	Ω -Based Consensus	457
17.4.1	Consensus from Alpha ₁ Objects and Ω	457
17.4.2	Consensus from an Alpha ₂ Object and Ω	459
17.4.3	Consensus from an Alpha ₃ Object and Ω	460
17.4.4	When the Eventual Leader Elected by Ω Does Not Participate	463
17.4.5	The Notion of an Indulgent Algorithm	464
17.4.6	Consensus Object Versus Ω	464
17.5	Wait-Free Implementations of the Alpha ₁ and Alpha ₂ Abstractions	465
17.5.1	Alpha ₁ from Atomic Registers	465
17.5.2	Alpha ₂ from Regular Registers	467
17.6	Wait-Free Implementations of the Alpha ₂ Abstraction from Shared Disks	472
17.6.1	Alpha ₂ from Unreliable Read/Write Disks	472
17.6.2	Alpha ₂ from Active Disks	476
17.7	Implementing Ω	477
17.7.1	The Additional Timing Assumption <i>EWB</i>	478
17.7.2	An <i>EWB</i> -Based Implementation of Ω	479
17.7.3	Proof of the Construction	481
17.7.4	Discussion	484

17.8	Summary	485
17.9	Bibliographic Notes	485
17.10	Exercises and Problems	486
Afterword		489
Bibliography		495
Index		509

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 $i \in \{1, \dots, m\}$ 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
% blabla %	Comments
;	Sequentiality operator between two statements

Figures and Algorithms

1.1	Operations to access a disk.	5
1.2	An interleaving of invocations to disk primitives.	6
1.3	Synchronization is to preserve invariants	8
1.4	Invariant expressed with control flows	8
1.5	Sequential specification of a lock object <i>LOCK</i>	12
2.1	An atomic register execution.	16
2.2	Peterson's algorithm for two processes: first component (code for p_i).	18
2.3	Peterson's algorithm for two processes: second component (code for p_i).	18
2.4	Peterson's algorithm for two processes (code for p_i)	20
2.5	Mutex property of Peterson's two-process algorithm (part 1)	21
2.6	Mutex property of Peterson's two-process algorithm (part 2)	21
2.7	Bounded bypass property of Peterson's two-process algorithm . . .	22
2.8	Peterson's algorithm for n processes (code for p_i)	22
2.9	Total order on read/write operations	24
2.10	A tournament tree for n processes.	27
2.11	Tournament-based mutex algorithm (code for p_i)	28
2.12	An n -process concurrency-abortable operation (code for p_i)	30
2.13	Access pattern to X and Y for a successful <code>conc_abort_op()</code> invocation by process p_i	32
2.14	Lamport's fast mutex algorithm (code for p_i)	33
2.15	Fischer's synchronous mutex algorithm (code for p_i)	37
2.16	Accesses to X by a process p_j	38
2.17	Test&set-based mutual exclusion	39
2.18	Swap-based mutual exclusion	40
2.19	Compare&swap-based mutual exclusion	41
2.20	From deadlock-freedom to starvation-freedom (code for p_i)	42
2.21	A possible case when going from deadlock-freedom to starvation-freedom	43

2.22	Fetch&add-based mutual exclusion	45
2.23	An execution of a regular register	46
2.24	An execution of a register	47
2.25	Lamport's bakery mutual exclusion algorithm	49
2.26	The two cases where p_j updates the safe register $FLAG[j]$	51
2.27	Aravind's mutual exclusion algorithm	54
2.28	Relevant time instants in Aravind's algorithm	55
3.1	From a sequential stack to a concurrent stack: structural view . . .	62
3.2	From a sequential to a concurrent stack (code for p_i)	63
3.3	Implementing a semaphore (code for p_i)	65
3.4	A semaphore-based implementation of a buffer	66
3.5	A production/consumption cycle	68
3.6	Behavior of the flags $FULL[x]$ and $EMPTY[x]$	69
3.7	An efficient semaphore-based implementation of a buffer	70
3.8	Blackboard and sleeping rooms	72
3.9	Resource allocation with priority (code for process p_i)	73
3.10	From a sequential file to a concurrent file	75
3.11	Readers-writers with weak priority to the readers	76
3.12	Readers-writers with strong priority to the readers	77
3.13	Readers-writers with priority to the writers	78
3.14	One writer and several readers from producer-consumer	79
3.15	Efficiency gain and mutual exclusion requirement	80
3.16	Several writers and several readers from producer-consumer	81
3.17	A register-based rendezvous object	83
3.18	A monitor-based rendezvous object	84
3.19	A simple single producer/single consumer monitor	85
3.20	Predicate transfer inside a monitor	86
3.21	Base objects to implement a monitor	87
3.22	Semaphore-based implementation of a monitor	88
3.23	A readers-writers monitor with strong priority to the readers	91
3.24	A readers-writers monitor with strong priority to the writers	92
3.25	The fairness properties P1 and P2	93
3.26	A readers-writers monitor with fairness properties	94
3.27	A monitor based on a scheduled wait operation	95
3.28	A buffer for a single producer and a single consumer	98
3.29	Operations $prio_down()$ and $prio_up()$	99
3.30	Derivation tree for a path expression	100
3.31	Control prefixes and suffixes automatically generated	101
3.32	A variant of a semaphore-based implementation of a buffer	103
3.33	Two buffer implementations	104
3.34	A readers-writers implementation	105
3.35	Railways example	106
3.36	Another readers-writers implementation	108

4.1	A sequential execution of a queue object	114
4.2	A concurrent execution of a queue object.	114
4.3	Structural view of a system.	116
4.4	Example of a history	119
4.5	Linearization points	123
4.6	Two ways of completing a history.	124
4.7	Atomicity allows objects to compose for free	127
4.8	A sequentially consistent history	129
4.9	Sequential consistency is not a local property	130
5.1	Interleaving at the implementation level.	137
5.2	Splitter object	141
5.3	Wait-free implementation of a splitter object (code for process p_i)	142
5.4	On the modification of <i>LAST</i>	143
5.5	Obstruction-free implementation of a timestamp object (code for p_i)	144
5.6	A typical use of compare&swap() by a process	145
5.7	The list implementing the queue	146
5.8	Initial state of the list.	147
5.9	Michael & Scott's non-blocking implementation of a queue	148
5.10	Shafiei's non-blocking atomic stack.	152
5.11	A simple wait-free implementation of an atomic stack.	153
5.12	On the linearization points of the wait-free stack.	154
5.13	Boosting obstruction-freedom	157
5.14	A contention-based enrichment of an obstruction-free implementation (code for p_i)	158
5.15	A contention manager to boost obstruction-freedom to non-blocking	159
5.16	A contention manager to boost obstruction-freedom to wait-freedom.	160
6.1	The initial state of the list.	168
6.2	Hybrid implementation of a concurrent set object	169
6.3	The remove() operation	170
6.4	The add() operation	170
6.5	The contain() operation.	170
6.6	A contention sensitive implementation of a binary consensus object	173
6.7	Proof of the contention sensitive consensus algorithm (a).	174
6.8	Proof of the contention sensitive consensus algorithm (b).	175
6.9	A double-ended queue	177
6.10	Definition of the atomic operations LL(), SC(), and VL() (code for process p_i)	178

6.11	Implementation of the operations <code>right_enqueue()</code> and <code>right_dequeue()</code> of a double-ended queue	179
6.12	How <code>DQ.right_enqueue()</code> enqueues a value	180
6.13	Examples of concurrency-free patterns	182
6.14	A concurrency-abortable non-blocking stack	182
6.15	From a concurrency-abortable object to a starvation-free object	183
7.1	A simple wait-free counter for n processes (code for p_i).	191
7.2	Wait-free weak counter (one-shot version, code for p_i)	194
7.3	Proof of the weak increment property	197
7.4	Fast read of a weak counter (code for process p_i)	199
7.5	Reading a weak counter (non-restricted version, code for process p_i)	200
7.6	A trivial implementation of a store-collect object (code for p_i)	203
7.7	A store-collect object has no sequential specification.	203
7.8	A complete binary tree to implement a store-collect object.	205
7.9	Structure of a vertex of the binary tree.	205
7.10	An adaptive implementation of a store-collect object (code for p_i)	207
7.11	Computing an upper bound on the number of marked vertices	211
7.12	Merging <code>store()</code> and <code>collect()</code> (code for process p_i)	212
7.13	Incorrect versus correct implementation of the store <code>collect()</code> operation	213
7.14	An efficient <code>store_collect()</code> algorithm (code for p_i).	214
7.15	Sequential and concurrent invocations of <code>store_collect()</code>	215
7.16	Concurrent invocations of <code>store_collect()</code>	216
8.1	Single-writer snapshot object for n processes	220
8.2	Multi-writer snapshot object with m components.	220
8.3	An obstruction-free implementation of a snapshot object (code for p_i)	221
8.4	Linearization point of an invocation of the <code>snapshot()</code> operation (case 1)	223
8.5	The <code>update()</code> operation includes an invocation of the <code>snapshot()</code> operation	224
8.6	Bounded wait-free implementation of a snapshot object (code for p_i)	225
8.7	Linearization point of an invocation of the <code>snapshot()</code> operation (case 2)	227
8.8	Single-writer atomic snapshot for infinitely many processes (code for p_i)	229
8.9	An array transmitted from an <code>update()</code> to a <code>snapshot()</code> operation	230

8.10	Wait-free implementation of a multi-writer snapshot object (code for p_i)	232
8.11	A snapshot() with two concurrent update() by the same process.	233
8.12	An execution of a one-shot snapshot object	239
8.13	An execution of an immediate one-shot snapshot object.	239
8.14	An algorithm for the operation update snapshot() (code for process p_i)	241
8.15	The levels of an immediate snapshot objects.	242
8.16	Recursive construction of a one-shot immediate snapshot object (code for process p_i)	245
9.1	Uncertainties for 2 processes after one communication exchange	253
9.2	A grid of splitters for renaming.	255
9.3	Moir-Anderson grid-based time-adaptive renaming (code for p_i)	255
9.4	A simple snapshot-based wait-free size-adaptive $(2p-1)$ -renaming implementation(code for p_i)	257
9.5	Recursive optimal size-adaptive renaming (code for p_i)	261
9.6	Recursive renaming: first, p_3 executes alone	262
9.7	Recursive renaming: p_1 and p_4 invoke new_name(4, 1, up)()	263
9.8	Borowsky and Gafni's recursive size-adaptive renaming algorithm (code for p_i)	267
9.9	Tree associated with a concurrent renaming execution	269
9.10	Simple perfect long-lived test&set-based renaming	270
10.1	An execution of a transaction-based two-process program	280
10.2	Execution of a transaction-based program: view of an external observer.	280
10.3	Structure of the execution of a transaction	283
10.4	Read from a consistent global state	285
10.5	Validation test for a transaction T	286
10.6	TL2 algorithms for an update transaction	287
10.7	TL2 algorithms for a read-only transaction	289
10.8	The list of versions associated with an application register X	290
10.9	JVSTM algorithm for an update transaction	291
10.10	JVSTM algorithm for a read-only transaction	293
10.11	Causal pasts of two aborted transactions.	294
10.12	An STM system guaranteeing the virtual world consistency condition	297
11.1	An execution of a regular register	307
11.2	An execution of a register	308

11.3	From SWSR safe/regular to SWMR safe/regular: a bounded construction (code for p_i)	311
11.4	A first counter-example to atomicity	312
11.5	SWMR binary register: from safe to regular	313
11.6	SWMR safe register: from binary domain to b -valued domain . . .	314
11.7	SWMR regular register: from binary domain to b -valued domain	315
11.8	A read invocation with concurrent write invocations	317
11.9	A second counter-example for atomicity.	319
11.10	SWMR atomic register: from bits to a b -valued register.	320
11.11	There is no new/old inversion	321
11.12	SWSR register: from regular to atomic (unbounded construction)	322
11.13	Atomic register: from one reader to multi-reader (unbounded construction)	325
11.14	Atomic register: from one writer to multi-writer (unbounded construction)	326
12.1	Two read invocations r and r' concurrent with an invocation w_{2i+1} of $R.write(1)$	333
12.2	A possible scenario of read/write invocations at the base level . . .	333
12.3	Tromp's construction of an atomic bit	336
12.4	What is forbidden by the properties A1 and A2	339
12.5	$\pi(r_1) \rightarrow_H w_2$	340
12.6	ρ_r belongs neither to r nor to r'	342
12.7	A new/old inversion on the regular register REG	343
13.1	Buffers and switch in Tromp's construction	349
13.2	Tromp's construction of a SWSR b -valued atomic register.	351
13.3	A write invocation concurrent with two read invocations	352
13.4	The write automaton of Tromp's construction.	353
13.5	The read automaton	353
13.6	Global state automaton.	354
13.7	Simultaneous accesses to the same buffer.	358
13.8	Successive read/write collisions.	360
13.9	Vidyasankar's construction of an SWSR b -valued atomic register	361
13.10	Successive updates of the atomic bit WR	362
13.11	Ordering on base operations (Lemma 30).	363
13.12	Overlapping invocations (atomicity in Theorem 56).	366
13.13	Vidyasankar's construction of an SWMR b -valued atomic register	367
14.1	From a sequential specification to a wait-free implementation . . .	372
14.2	Architecture of the universal construction.	376

14.3	A wait-free universal construction (code for process p_i)	379
14.4	The object Z implemented as a linked list	385
14.5	Herlihy's bounded wait-free universal construction	386
14.6	Sequence numbers	389
14.7	Multi-valued consensus from binary consensus: construction 1 . . .	393
14.8	Multi-valued consensus from binary consensus: construction 2 . . .	394
14.9	Linearization order for the proof of the termination property	395
15.1	t -Tolerant self-implementation of an object RO	400
15.2	t -Tolerant SWSR atomic register: unbounded self-implementation (responsive crash)	401
15.3	t -Tolerant SWSR atomic register: bounded self-implementation (responsive crash)	403
15.4	Order in which the operations access the base registers	404
15.5	Proof of the "no new/old inversion" property	405
15.6	A simple improvement of the bounded construction	406
15.7	t -Tolerant SWSR atomic register: unbounded self-implementation (non-responsive crash)	407
15.8	Wait-free t -tolerant self-implementation of a consensus object (responsive crash/omission)	409
15.9	Wait-free t -tolerant (and gracefully degrading) self-implementation of an SWSR saferegister (responsive arbitrary failures)	412
15.10	Gracefully degrading self-implementation of a consensus object (responsive omission)	415
16.1	Existence of a bivalent initial configuration	424
16.2	Read/write invocations on distinct registers	426
16.3	Read and write invocations on the same register	427
16.4	From test&set to consensus (code for p_i , $i \in \{0, 1\}$)	430
16.5	From an atomic concurrent queue to consensus (code for p_i , $i \in \{0, 1\}$)	431
16.6	From a swap register to consensus (code for p_i , $i \in \{0, 1\}$)	432
16.7	$Q.enqueue()$ invocations by the processes p and q	434
16.8	State of the atomic queue Q in configuration $q(p(D))$	435
16.9	Assuming that S_p contains at most k invocations of $Q.dequeue()$	436
16.10	Assuming that S_q does not contain invocations of $Q.dequeue()$	436
16.11	From the configuration D to the configuration D_0 or D_1	438
16.12	From compare&swap to consensus	439
16.13	From mem-to-mem-swap to consensus (code for process p_i)	440
16.14	From an augmented queue to consensus	442
16.15	From a sticky bit to binary consensus	443

17.1	From compare&swap to alpha and omega	451
17.2	Obligation property of an α_2 object	455
17.3	From α_1 (adopt-commit) objects and Ω to consensus	458
17.4	From an α_2 object and Ω to consensus	460
17.5	From an α_3 (store-collect) object and Ω to consensus	461
17.6	Wait-free implementation of <code>adopt_commit()</code>	466
17.7	Timing on the accesses to <i>AA</i> for the proof of the quasi-agreement property	466
17.8	Timing on the accesses to <i>BB</i> for the proof of the quasi-agreement property	467
17.9	Array of regular registers implementing an α_2 object	468
17.10	Wait-free construction of an α_2 object from regular registers	469
17.11	Regular register: read and write ordering	471
17.12	Replicating and distributing <i>REG</i> [1.. <i>n</i>] on the <i>m</i> disks	472
17.13	Implementing an SWMR regular register from unreliable read/write disks	473
17.14	Wait-free construction of an α_2 object from unreliable read/write disks	475
17.15	The operations of an active disk	476
17.16	Wait-free construction of an α_2 object from an active disk . . .	477
17.17	A <i>t</i> -resilient construction of Ω (code for <i>p_i</i>)	481
17.18	The operations <code>collect()</code> and <code>deposit()</code> on a closing set object (code for process <i>p_i</i>)	487

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: $\text{seek}(x)$, which moves the disk read/write head to the address x ; $\text{read}()$, which returns the value located at the current position of the read/write head; and $\text{write}(v)$, 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 $\text{disk_read}(x)$ 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 $\text{disk_write}(x, v)$ described in the same figure.

The disk primitives $\text{seek}()$, $\text{read}()$, and $\text{write}()$ 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 $\text{disk_read}(x)$ and later (after p 's invocation has terminated) another process q invokes $\text{disk_write}(y, v)$, 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 %
     $D.\text{seek}(x); r \leftarrow D.\text{read}(); \text{return}(r)$ 
end operation.

operation disk_write( $x, v$ ) is
     $D.\text{seek}(x); D.\text{write}(v); \text{return}()$ 
end operation.

```

Fig. 1.1 Operations to access a disk

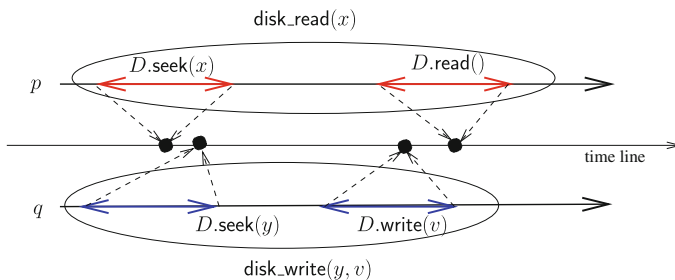


Fig. 1.2 An interleaving of invocations to disk primitives

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

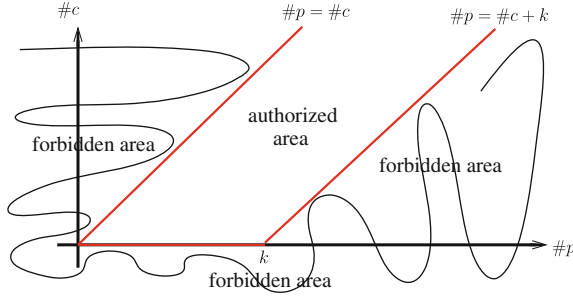


Fig. 1.3 Synchronization is to preserve invariants

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 i th data item production and the j th 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.

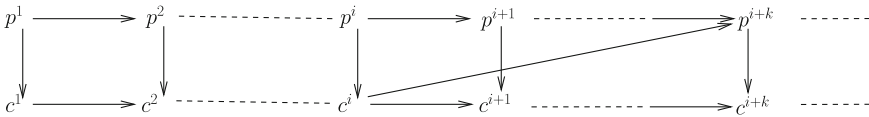


Fig. 1.4 Invariant expressed with control flows

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 $A, B, C \dots$ (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 $(LOCK.acquire_lock(); LOCK.release_lock())^*$ (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.

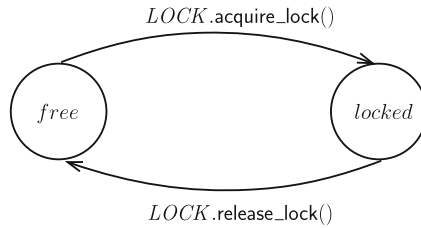


Fig. 1.5 Sequential specification of a lock object *LOCK*

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].