Algorithms for Mutual Exclusion

M. Raynal

Translated by
D. Beeson

Algorithms for Mutual Exclusion

M. Raynal

Translated by
D. Beeson

The MIT Press
Cambridge, Massachusetts

First MIT Press edition, 1986

Original edition published under the title *Algorithmique du parallélisme* by Dunod informatique, France. © 1984 Bordas, Paris.

Published in Great Britain by
NORTH OXFORD ACADEMIC Publishers Limited
a subsidiary of Kogan Page Limited
120 Pentonville Road
London N1 9JN

Printed in Great Britain

Library of Congress Cataloging in Publication Data
Raynal, M.
Algorithms for mutual exclusion.

Translation of: Algorithmique du parallélisme.
Bibliography: p.
Includes index.
1. Parallel processing (Electronic computers)
2. Electronic data processing – Distributed processing.
3. Algorithms. I. Title.
QA76.5.R38513 1986 001.64 85-7916

ISBN 0-262-18119-3

Series Foreword

It is often the case that the periods of rapid evolution in the physical sciences occur when there is a timely confluence of technological advances and improved experimental technique. Many physicists, computer scientists and mathematicians have said that such a period of rapid change is now under way. We are currently undergoing a radical transformation in the way we view the boundaries of experimental science. It has become increasingly clear that the use of large-scale computation and mathematical modeling is now one of the most important tools in the scientific and engineering laboratory. We have passed the point of viewing the computer as a device for tabulating and correlating experimental data; we now regard it as a primary vehicle for testing theories for which no practical experimental apparatus can be built. For example, NASA scientists speak of 'numerical' wind tunnels, and physicists experiment with the large-scale structure of the universe completely by computer simulation.

The major technological change accompanying this new view of experimental science is a blossoming of new approaches to computer architecture and algorithm design. By exploiting the natural parallelism in scientific applications, new computer designs show the promise of major advances in processing power. When coupled with the current biennial doubling of memory capacity, supercomputers are on their way to becoming the laboratories of much of modern science. In addition, we are seeing a major trend toward distributing operating systems over a network of a powerful and perhaps inhomogenous, set of processors. The idea of being able to make diverse computing resources cooperate on the solution of a large problem is most exciting to the scientist whose application is part numerical, part symbolic, manages a massive data base, and outputs complex graphics.

In this series we hope to focus on the effect these changes are having on the design of both scientific and systems software. In particular, we plan to highlight many major new trends in the algorithms and the associated programming and software tools that are being driven by the new advances in computer architecture. Of course, the relation between algorithm design and computer architecture is symbiotic. New views on the structure of physical processes demand new computational models,

which then drive the design of new machines. We can expect progress in this area for many years, as our understanding of the emerging science of concurrent computation deepens.

As most computer scientists realize, distributed, concurrent hardware systems represent the future of our interaction with the computers. The contemporary vision of having a thousand processors and work stations networked into a system with vast cumulative resources has captivated both industry and academia. Unfortunately, there is much more to making this a reality than just hooking the hardware together. Problems such as the synchronization of distributed concurrency, distribution of work among processors, and mantaining the integrity of a dynamic, distributed file system are numerous enough to keep a generation of scientists busy.

In *Algorithms for Mutual Exclusion*, Michel Raynal presents a unified view of the algorithms associated with mutual exclusion in concurrent systems. He considers both classical shared memory systems as well as network-based concurrency. The problems and solutions presented in this book must become part of the working knowledge of anyone seriously interested in building concurrent systems.

Dennis B. Gannon

Foreword

Constant progress in technology (in particular microprocessors and local area networks) and in programming methodology (using languages of the Ada type) is increasingly opening up computer systems as a field of research, traditionally the preserve of 'initiates', to applications designers and implementors. There are many problems involved in the design of such systems, such as the management of common memories and memories local to the various processors, the allocation of physical and virtual resources defined within the system, and concurrency protection and management. It is primarily issues of concurrency that are becoming increasingly critical. Mechanisms for the management of processes and their concurrency must often be implemented within a genuinely parallel framework, as in multiprocessor systems with or without common memory.

There is one fundamental problem that stands out from all those involved in controlling parallelism: mutual exclusion. The issue here is one of ensuring that it is possible, given a number of parallel programs, to limit their parallelism at certain points in their execution: as one program enters a particular zone of code it must exclude the others. It is this question, mutual exclusion, and its expression in algorithms that we shall consider in this book. We shall not therefore be concerned with systems design as a whole (the interested reader should consult works such as 'Crocus', *'Systèmes d'exploitation des ordinateurs'*, or 'Cornafion', *'Systèmes informatiques répartis'*) but with a particular systems component which is becoming increasingly important in their definition.

This book is divided into four parts. The first (Chapter 1) introduces the problems associated with controlling parallelism, following a deliberately didactic approach; we shall discuss mutual exclusion, deadlock and data coherence in turn. The second part (Chapters 2 and 3) describes algorithms (Chapter 2) and statements (Chapter 3) to implement mutual exclusion in a centralized framework (i.e. via access to a common memory). The third part (Chapters 4 and 5) deals with distributed algorithms for mutual exclusion. Chapter 4 will be concerned with solutions based on state variables, and Chapter 5 will describe algorithms based on message communication, the kind of protocols needed for implementation of exclusion on a network. The final

part (Chapter 6) will consider original software approaches to two control problems. Although there are some cross-references, chapters can be read independently of each other.

This book is intended for computer scientists interested in the design, construction and implementation of centralized or distributed computer systems, where by 'computer systems' we mean systems as different as operating systems, database systems, document or business management systems, real-time systems, process control systems, etc. It should be of value to engineers, who will find that it contains an inventory of algorithms which are generally scattered through the specialist literature, and are sometimes difficult to track down. All the algorithms are presented in the same way: underlying principles and assumptions, the algorithm itself, proof of its valid operation with respect to expected behaviour, and comments on its structure and efficiency. From this point of view, the book is a collection of algorithms on a particular subject, of a kind that is common, for sequential programming, in works on sorting, automata and fundamental data structure. The book is also intended for students at Masters level and above, for trainee engineers in computer science and for researchers who would like to become more closely acquainted with the systems design field. Teachers will find that it contains useful supplementary material, because of the approach and the analytic point of view adopted, to standard survey or design textbooks, and could treat it as a collection of exercises on the subject of mutual exclusion. This is, in fact, a textbook on algorithms for parallel processing.

Any corrections pointed out by readers to errors we may have made will be gratefully received.

Description language and notation

The language used to describe these algorithms is a 'Pascal-style' language. We shall be essentially concerned with the basic data types: integers, scalars and Booleans; and traditional control structures: conditions, iterations, etc., where the end of a statement is explicitly given mr ***if*** ... ***endif***, ***do*** ... ***enddo***, etc. Any possible ambiguities in the interpretation of control structures are avoided by using the construction ***begin*** ... ***end*** to delimit processes. Assignment is shown using the sign ←.

An active delay is expressed using loops in which the statement ***nothing*** is used for the activity associated with waiting. The statement ***wait*** is used for waiting whether it is passive or active. In the latter case it is a shorthand form for which the semantics is:

wait *C* = ***while*** ¬ *C* ***do nothing enddo***

The general format used to describe algorithms is as follows:

< *prelude* >;
< *critical section* >;
< *postlude* >;

the < *prelude* > and < *postlude* > sections make up the protocol that processes must follow to enter and leave the critical section. Those parts of a process that do not include the critical section or the protocol make up the non-critical part.

Set and logical notation will sometimes be used to make writing easier. On sets we therefore have the following equivalent notations:

$$\textit{for } i \neq j,\ i \in 1..n \textit{ do } .. \equiv \textit{for } i \textit{ from } 1 \textit{ to } j-1, \textit{ from } j+1 \textit{ to } n \textit{ do } \ldots$$

$$\textit{wait } (\forall\ i \in 1..n : a) \equiv$$
$$\textit{wait } (a,1 \wedge a,2 \wedge \ldots \wedge a,n)$$

We shall use conventional symbols with Boolean expressions:

$\neg$: not
$\vee$: or
$\wedge$: and
$\forall$: for all
$\exists$: there exists

Over the past 20 years, the 'mutual exclusion phenomenon' has emerged as one of the best paradigms of the difficulties associated with parallel or distributed programming.

It must be stressed at the outset that the implementation of a mutual exclusion mechanism is a very real task facing every designer of operating systems.

Even applications programmers must take a certain interest in the question, because they will have to use services provided by computer systems built around several processing units, or even several computers linked by a network. The problem is simple enough to state: what we have to do is to define fundamental operations that make it possible to resolve conflicts resulting from several concurrent processes sharing the resources of a computer system. The whole complex structure of synchronization and communication between the elements of a parallel or distributed system depends on the existence of such operations. Algorithms solving this problem are generally made up of only a dozen or so lines of code and contain only trivial assignment instructions. Parallelism, however, makes it difficult to understand their behaviour and to analyse their properties, such as avoidance of deadlock or fair conflict resolutions.

Given both the practical importance and the inherent difficulty of the problem, it is not at all surprising that a vast number of works have been published on the subject. What Michel Raynal offers here is the most important results of all this research.

Michel Raynal's aim is not merely to produce a catalogue of the various algorithms found in scientific journals or conference papers, but to provide us with a remarkable survey of the field. All the algorithms have been rewritten in a single language and restructured so as to make them easy to understand and compare. The presentation of these algorithms systematically stresses the principles guiding their design, provides arguments to prove their validity and gives quantitative data allowing their assessment. This is, as far as we know, a unique book on the subject, opening up a vast field of research which is both firmly based and highly complex: algorithms for parallel or distributed control.

There is no doubt that this is a work that must be regarded as indispensable in the library of teachers, researchers or engineers working in this field, who will use it to illustrate a lecture, launch new research or solve specific concrete problems.

Gérard Roucairol
(University of Paris-Sud)

Contents

Chapter 1

The nature of control problems in parallel processing

1.1. Processes and their interactions

The need for the control of a set of parallel processes is a consequence of the many problems associated with their management. Responsibility for this control is left to what is generally called a *system*: an operating system, a data base management system, a real time system, etc., depending on the purpose for which it is constructed.

The concept of a process is used both to express the activity associated with users of the system and the internal structure of the system itself. A number of educational works such as Crocus (1975), Shaw (1974), Peterson and Silberschatz (1983), or Lister (1979), discuss the concept and make its importance clear, illustrating it by means of examples.

In this chapter we shall start by specifying the interactions within a set of parallel processes (Section 1.1), after which we shall examine the nature of problems associated with these interactions (Section 1.2). The search for solutions to these problems has led to the emergence of algorithms of a new kind, which we call algorithms for parallel processing. The rest of this work is devoted to a study of these algorithms from the point of view of mutual exclusion.

1.1.1. The process concept

The process concept was first introduced into information systems in order to highlight the differences between a program as text (written in a certain language) and the execution of this program on a processor. Since then, the concept has been the subject of much study, leading to its more formal definition (Horning and Randell 1973). A major development was its introduction into programming languages (Wulf et al. 1971). This occurred in response to two factors. First there was the influence of technology. The concept of a process to express the idea of an activity (and which can therefore be used to structure the overall activity of a system formed from elementary activities corresponding to the different parts of the machine) has become an indispensable tool with the appearance of multiprocessors and computer

networks (whatever their size). If we are to make satisfactory use of such machines, it is absolutely necessary to be able to master the different activities taking place within them.

At the same time, research into programming methodology (structured programming (Dijkstra 1968, Dahl et al. 1972, Hoare 1972, Brinch Hansen 1973), programming theory, etc., showed that in order to solve a problem it was necessary to adopt an approach based on successive refinements and which applied the principle of abstraction, by which we mean that at a given level we consider only the function offered at that level independently of the way in which it is implemented. This research led to the introduction into programming languages of the concepts of processes and abstract types. The former, as we have already pointed out, is the expression within the language of the idea of an activity [i.e. of an active object (Brinch Hansen 1975, 1978, Hoare 1978)] ; the latter expresses the idea of data that the program can manipulate [i.e. the idea of a passive object (Hoare 1974, Liskov et al. 1977, Wirth 1977)].

In this way processes gradually became familiar objects to all program designers: a program would no longer always be seen (implicitly) as a single process, but could be explicitly expressed as a set of processes whatever the purpose of the program, whether implementing a system (i.e. an interface between a machine and users) or a particular application.

It is interesting that the concept of a coroutine, which makes it possible to decompose a program into a certain number of processes of which only one is active at any given moment, was introduced as early as 1963 to deal with a particular class of applications: compilers structured as several processes communicating data to each other in a 'pipeline' fashion (Conway 1963). As at that time the concept of a process did not exist in programming languages, it was up to the programmer to use both a given language and the tools provided by the system to assist in the production of software to obtain the desired behaviour. The introduction of this concept into a language makes it possible to leave all this work to the compiler; the program designer need then only concern himself with the expression of a solution to his problem in the language in question.

In the rest of our discussion we shall assume the principle that a program should be structured in terms of parallel processes whatever the techniques used for producing the program: parallel languages or sequential languages plus production tools. A fundamental question is then immediately raised: what are the interactions between processes themselves?

1.1.2. The criterion of mutual awareness

In order to reply to this question we shall consider two criteria. Like any criteria designed to distinguish absolutely between classes, ours will be to some extent arbitrary; their value lies in the 'didactic' classification that they allow for the various problems associated with control. The criteria are independent

of the number of processes n (where $n > 2$) and of the level at which these processes are viewed (applications, systems).

The first criterion concerns the extent to which a process is aware of the environment (made up of other processes) with which it interacts:

Criterion 1: What degree of awareness do processes have of each other?

This criterion allows us to define two classes of processes:

C_1: processes unaware of each other

C_2: processes aware of each other.

The second class may itself be broken down into two subclasses:

C_{21}: processes indirectly aware of each other (e.g. because they use a shared object)

C_{22}: processes explicitly aware of each other (and which therefore have communication primitives available to them).

These different degrees of awareness that processes have of each other lead to different relationships between them:

R_1: competition

R_{21}: cooperation by sharing

R_{22}: cooperation by communication.

COMPETITION

In this case the processes are totally unaware of the existence of other processes: they come into conflict for the use of objects which they must leave in the same state as they found them, precisely because, as each is unaware of the existence of the others, if an object is unique, it must be the same for all of them. The objects involved in such conflicts generally make up what is known as the system resources. A *resource* is usually a model of a device (memory, peripheral, CPU, clock, etc.). The objects are not modified by the processes but they are indispensable to their operation — their role is analagous to catalysts in chemistry, as opposed to substances that undergo transformation; as is the case of catalysts when writing chemical equations, resources do not generally appear in the statement expressing a process. Synchronization rules aimed at resolving the problems associated with physical constraints must be defined so that competition between processes can take place without leading to difficulties.

Comment

Competitive interactions affect all processes executing on a given computer. To avoid them altogether it is necessary to provide more resources than there

are processes that might need them at any given time, so that they can be allocated in such a way as to avoid conflict. However, this number is unknown and it is impossible to create physical resources dynamically. Competitive relationships are therefore the 'minimal' relationships between processes.

COOPERATION BY SHARING

Here, processes interacting with each other know there are other processes, without being explicitly aware of them. This is what happens, for example, with variables shared between different processes in the system or a data base: processes (or transactions) use and update the data base without reference to other processes, but know that they might be using or updating the same data.

Two read operations, for example, on a single data base may lead to different results, depending on whether or not another process has been writing to the data base. We are no longer dealing with resources that have to be allocated, but with shared data that the processes may transform. The processes must cooperate in ensuring that the data base they share is properly managed. We are dealing here with data-oriented systems, and the control mechanisms implemented must ensure that the data shared remains coherent.

Comment

Cooperative and competitive interactions between processes are by no means mutually exclusive. On the contrary: shared data by way of which processes cooperate with each other are held in resources over which the same processes will come into conflict. Synchronization rules must therefore not only sort out conflicts of access but also guarantee the coherence of the data accessed.

Within the terms of a single problem for which a layered solution has been proposed using processes, cooperative interactions at any one level may well correspond to competitive interactions at another level and vice versa. This contradicts nothing in what we are saying: we are concerned with classifying types and relationships independently of the level at which we are making our observations.

COOPERATION BY COMMUNICATION

In the first two cases above, the environment of a process does not contain the other processes, and interactions between them are always indirect. In the first case they were sharing devices without knowing it, and in the second they were sharing values. In this case, however, every process has an environment that contains names of other processes with which it may explicitly exchange data. Each process no longer has its own particular aim, but participates in a common goal which links the whole set of processes. We are dealing here with

what are known as message systems, or systems of communicating processes; these systems are characterized by the presence of message transmission and reception primitives. These primitives may be provided by languages containing the appropriate linguistic constructions in their definitions [e.g. CSP (Hoare 1978) or Ada (Ichbiah et al. 1983)], they may be constructed by the programmer in a language that does not itself contain them but which does make tools available for their construction [e.g. concurrent Pascal (Brinch and Hansen 1975)], or they may be provided by a system kernel accessible to applications programs.

Comment

Access primitives and the rules governing their use are affected by the distinction we have drawn between cooperation by sharing and by communication. In cooperation by sharing, processes are unaware of each other and their cooperation primitives are the operations *read* and *write* defined on an object, and the access rule may be formally described by:

$$(read + write)^*$$

In cooperation by communication, the cooperation primitives provided are the operations *send* and *receive*, and for a pair of communicating processes we have a communication rule which we can formally describe by:

$$(send\,.\,receive)^*$$

There are cases (languages or systems) where processes are not explicitly aware of each other and use the *send* and *receive* operations all the same; under such circumstances these operations are addressed to an object of the port type (Balzer 1971, Mitchell et al. 1978) or of the communication channel type (Ambler et al. 1977). As it is not our aim to produce a classification of languages (the interested reader should consult Le Guernic and Raynal 1980), we shall only consider the two major cooperation classes (sharing/communication), and these cases obviously belong in the latter class.

1.1.3. The corresponding criterion of mutual influence

Having used the criterion of mutual awareness to make a first, rough division of the interactions between processes into two major classes — competitive and cooperative — we can now attempt to refine our analysis by asking a second question, which will allow us to come more closely to grips with the problems posed by these kinds of relations between processes.

Criterion 2: *When processes interact competitively or cooperatively, what is the influence that the behaviour of one will have on the behaviour of others?*

In the case of competition, there is no exchange of information between processes — each has its own code and consequently the results of one process cannot be affected by the actions of others. On the other hand, the behaviour of one may be affected by the other: if there is competition between two processes for a single resource, then one will have to wait for the other to finish before using the resource. Thus, one process will have been slowed down by the other. Ultimately, it is possible for a process to be denied access to the resource indefinitely, in which case it would never terminate, and would give no result (a situation similar to that of a sequential program trapped in a loop). With cooperative interactions, a process may directly infuence another's results by means of the exchange of information.

In other words, the two cases may be distinguished by the extent to which the partial correctness of a process is independent of that of the others.

1.2. Control problems

In this section we shall examine the various problems associated with interactions between processes. To do so, we will use simple models which make the problems easy to grasp.

1.2.1. Competition between *n* processes for one resource

Consider the case of n processes in conflict for access to a single non-sharable resource. As the resource can only be used by a single process at a time, we shall call it a *critical resource* and it will be used in a *critical section* of the process. It will therefore be necessary to bracket the use of this resource by a protocol made up of two parts, concerned with acquiring and releasing the resource, which ensure that the resource R is used by only one process P_i:

acquisition protocol	*written*: (*R*
< *use of the resource in the critical section* >	*CS*
release protocol	)*R*

This protocol guarantees what is generally known as mutual exclusion: if, while the resource is being used by a process P_i, another process executes its acquisition protocol, that protocol must delay it until P_i has executed its release protocol.

Defining protocols is a complex problem — there are a number of pitfalls to avoid.

The first pitfall involves what is called *deadlock*. Consider several processes all attempting to enter their critical section to use the resource. As at most only one process may be in the critical section, one 'solution' would be to let none of them in. This is frequently the case when several people meet before a doorway (the resource). Imagine that they all follow the protocol: if I am