Prentice Hall International Series in Computer Science

C. A. R. Hoare, Series Editor

BACKHOUSE, R. C., *Program Construction and Verification*
BACKHOUSE, R. C., *Syntax of Programming Languages: Theory and practice*
DE BAKKER, J. W., *Mathematical Theory of Program Correctness*
BIRD, R., and WADLER, P., *Introduction to Functional Programming*
BJÖRNER, D., and JONES, C. B., *Formal Specification and Software Development*
BORNAT, R., *Programming from First Principles*
BUSTARD, D., ELDER, J., and WELSH, J., *Concurrent Program Structures*
CLARK, K. L., and McCABE, F. G., *micro-Prolog: Programming in logic*
CROOKES, D., *Introduction to Programming in Prolog*
DROMEY, R. G., *How to Solve it by Computer*
DUNCAN, F., *Microprocessor Programming and Software Development*
ELDER, J., *Construction of Data Processing Software*
ELLIOTT, R. J., and HOARE, C. A. R., (eds.), *Scientific Applications of Multiprocessors*
GOLDSCHLAGER, L., and LISTER, A., *Computer Science: A modern introduction (2nd edn)*
GORDON, M. J. C., *Programming Language Theory and its Implementation*
HAYES, I. (ed.), *Specification Case Studies*
HEHNER, E. C. R., *The Logic of Programming*
HENDERSON, P., *Functional Programming: Application and implementation*
HOARE, C. A. R., *Communicating Sequential Processes*
HOARE, C. A. R., and JONES, C. B. (ed.), *Essays in Computing Science*
HOARE, C. A. R., and SHEPHERDSON, J. C. (eds.), *Mathematical Logic and Programming Languages*
HUGHES, J. G., *Database Technology: A software engineering approach*
INMOS LTD, *occam Programming Manual*
INMOS LTD, *occam 2 Reference Manual*
JACKSON, M. A., *System Development*
JOHNSTON, H., *Learning to Program*
JONES, C. B., *Software Development: A rigorous approach (OOP)*
JONES, C. B., *Systematic Software Development using VDM (2nd edn)*
JONES, C. B. and SHAW, R. C. F. (eds.), *Case Studies in Systematic Software Development*
JONES, G., *Programming in occam*
JONES, G., and GOLDSMITH, M., *Programming in occam 2*
JOSEPH, M., PRASAD, V. R., and NATARAJAN, N., *A Multiprocessor Operating System*
LEW, A., *Computer Science: A mathematical introduction*
MACCALLUM, I., *Pascal for the Apple*
MACCALLUM, I., *UCSD Pascal for the IBM PC*
MARTIN, J. J., *Data Types and Data Structures*
MEYER, B., *Introduction to the Theory of Programming Languages*
MEYER, B., *Object-oriented Software Construction*
MILNER, R., *Communication and Concurrency*
MORGAN, C., *Programming from Specifications*
PEYTON JONES, S. L., *The Implementation of Functional Programming Languages*
POMBERGER, G., *Software Engineering and Modula-2*
REYNOLDS, J. C., *The Craft of Programming*
RYDEHEARD, D. E., and BURSTALL, R. M., *Computational Category Theory*
SLOMAN, M., and KRAMER, J., *Distributed Systems and Computer Networks*
SPIVEY, J. M., *The Z Notation: A reference manual*
TENNENT, R. D., *Principles of Programming Languages*
WATT, D. A., *Programming Language Concepts and Paradigms*
WATT, D. A., WICHMANN, B. A., and FINDLAY, W., *ADA: Language and methodology*
WELSH, J., and ELDER, J., *Introduction to Modula-2*
WELSH, J., and ELDER, J., *Introduction to Pascal (3rd edn)*
WELSH, J., ELDER, J., and BUSTARD, D., *Sequential Program Structures*
WELSH, J., and HAY, A., *A Model Implementation of Standard Pascal*
WELSH, J., and McKEAG, M., *Structured System Programming*
WIKSTRÖM, Å., *Functional Programming using Standard ML*

Principles of Concurrent and Distributed Programming

M. BEN-ARI
Technion – Israel Institute of Technology

PRENTICE HALL
New York London Toronto Sydney Tokyo Singapore

First published 1990 by
Prentice Hall International (UK) Ltd
66 Wood Lane End, Hemel Hempstead
Hertfordshire, HP2 4RG
A division of
Simon & Schuster International Group

Printed and bound in Great Britain at the
University Press, Cambridge

Library of Congress Cataloging-in-Publication Data

Ben-Ari, M., 1948-
Principles of concurrent and distributed programming / M. Ben-Ari.
p. cm. — (Prentice Hall international series in computer science)
Includes bibliographical references.
ISBN 0-13-711821-X : $31.95
1. Parallel processing (Electronic computers) 2. Electronic data processing—Distributed processing. I. Title. II. Series.
QA76.5.B393 1990
004'.36—dc20 89-39815
CIP

British Library Cataloguing in Publication Data

Ben-Ari, M., *1948-*
Principles of concurrent and distributed programming. —
(Prentice Hall international series in computing science)
1. Computer systems. Operating systems. Concurrent
I. Title.
005.4'2

ISBN 0-13-711821-X

5 94 93

1 [1]

Contents

Preface

The field of concurrent programming has seen an explosive expansion since the publication of my previous book *Principles of Concurrent Programming*. Two factors have dictated the need for an entirely new text: the increasing importance of distributed computing and the routine use in industry of languages containing primitives for concurrency such as Ada and occam. The aim of the book remains unchanged: it is an introductory textbook on concurrent programming that focuses on general principles and not on specific systems. The student who masters the material will be prepared not only to read the research literature, but also to evaluate systems, algorithms and languages from a broad perspective.

The intended audience includes advanced undergraduate and beginning graduate students, as well as practicing software engineers interested in obtaining a scientific background in this field. The book assumes a high level of 'computer maturity', such as would be achieved by 2–3 years of a computer science major or practical experience. At the very least, courses in data structures, programming languages and computer architecture should be required prerequisites.

The scope of the material has been limited to concurrent execution of processes, excluding the very low-grained concurrency of large parallel computers and at the other extreme the very high-grained concurrency of computer networks. Within this scope, the choice of material has been guided by pedagogical considerations. An algorithm, language or system is described because it demonstrates some fundamental concept or unusual behavior. Pointers to the current state of the art are given in the references.

The book is divided into three parts and a set of appendices. Part I covers classical concurrent programming similar to the treatment in *Principles of Concurrent Programming*, though there is more use of formal notation. Part II covers distributed programming. Three very different languages are described and compared: Ada, occam and Linda. This is followed by several chapters on distributed algorithms. Part III discusses *principles* of the the implementation of concurrency. A unique feature is the treatment of concurrent programming in real-time systems.

The Ada programming language is used uniformly throughout the book. Ada is probably the only language with embedded concurrent primitives for which high-quality implementations are available on a wide range of computers from

impression of speed given by a 200 characters-per-second printer. Yet these rates are extremely slow compared with the time required by the computer to process each character which may be about 10 microseconds.

Let us multiply the time scale by one million so that every microsecond becomes a second. Then every second becomes 11.5 days. If you were a computer, processing the characters being typed in, you would have to do 10 seconds of work every 2.3 days. Even if you had to process characters being sent to the printer, it would require only 10 seconds of work every 1.4 hours – hardly a strenuous job.

The tremendous gap between the speeds of human and mechanical processing on the one hand, and the speed of electronic devices on the other led to the development of operating systems which allow I/O operations to proceed 'in parallel' with computation. Obviously, controlling I/O (getting a character from the keyboard and placing it in memory) cannot be done in parallel with other computation on a single computer, but it is possible to 'steal' the few microseconds needed to control the I/O from the main computation. As can be seen from the numbers in the previous paragraph, the degradation in performance will not be noticeable, even when the overhead of switching is included.

What is the connection between overlapped I/O and concurrency? It would be theoretically possible for every program to include code that would periodically sample the keyboard and the printer to see if they need to be serviced. Alternatively, an I/O device could be instructed to cause the processor to jump to a pre-defined place in the program whenever it needs processing. However, these techniques require that every programmer include code to control every I/O device on the computer.

It is conceptually simpler to program the I/O controllers as separate *processes* which are executed in parallel with the main computation process (Figure 1.1). Of course we all know that on a single processor, the processes will not be executed in parallel, but the concurrent programming abstraction allows us to write independent programs for each device and free the application programmer from the details of the system. The techniques described in this book show how to *synchronize* the execution of the processes and how to *communicate* data among them.

1.3 Multi-programming

It is not too difficult to construct an operating system that overlaps I/O and computation. It is more difficult to write an applications program that takes advantage of this. Most programs are written to loop through a cycle: `Read; Process; Write` and it is not easy to design a program that correctly manages the data structures needed for concurrent I/O.

A simple generalization of the overlapped I/O within a single program is to overlap the computation and I/O of several programs (Figure 1.2). Each program can execute a simple `Read; Process; Write` cycle and the I/O of one is overlapped with the computation of another. *Multi-programming* is the concurrent execution of several independent programs on one processor.

An additional generalization is *time-slicing* – sharing the processor among

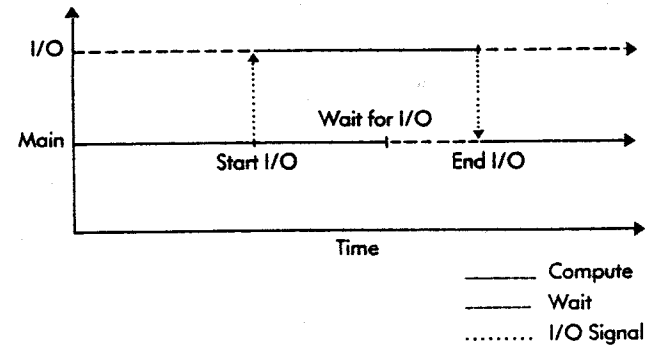

Figure 1.1 Overlapped I/O and computation

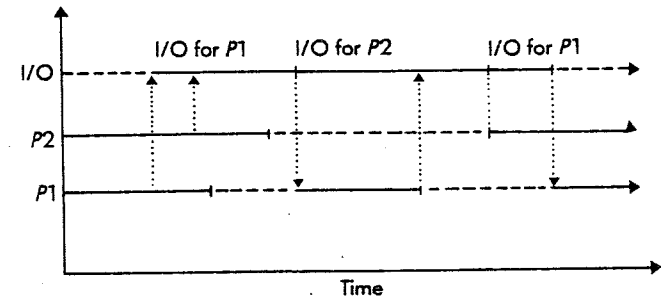

Figure 1.2 Multi-programming

several computations. Rather than wait for a computation to block pending completion of an I/O operation, a (hardware) timer is used to interrupt a computation at pre-determined intervals, for example every half second. A *scheduler* program is run by the operating system to determine which process should be allowed to run for the next interval. The scheduler can take into account priority considerations.

Interactive time-sharing systems use time-sliced multi-programming to give a group of users the illusion that each one has access to a dedicated computer. Of course, the computer will have to be much more powerful than one that could support only a single user so that the *response time* to a request remains acceptable. The fluctuation in the computing requirements of a group of users allows a time-sharing system to give good service while achieving economies of scale. Of course, if the system is overloaded with too many users, the response times will become unacceptable. A further advantage of a time-sharing system is that large computations can be run at a low priority to 'soak up' unused computing power and they can run during periods of light use such as nights and weekends.

Multi-programming can even be useful on a personal computer or workstation where all the work is being done for one user. In a typical configuration, several processes will be concurrently active, each with its own *window* on the screen for interaction with the user (Figure 1.3). These are:

- A background calculation on a spreadsheet.
- A word processor for writing a memo.
- A communications program receiving mail from a network.

The key to the successful use of multi-programming in a single-user environment is to ensure that concurrently active programs use different resources: the processor, the disk, the display, etc. If we had two processes using the processor intensively, we would pay for the overhead of concurrency and experience degraded performance with no apparent advantage.

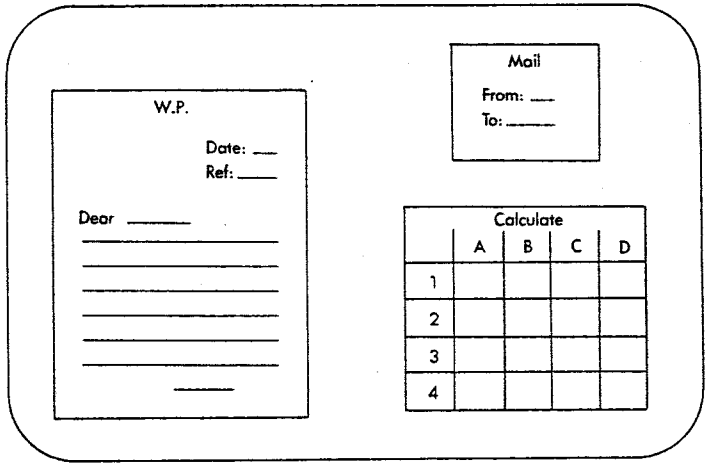

Figure 1.3 Windows on a workstation

Clearly, the concurrent programming abstraction is applicable to multi-programming. We cannot build one program that computes what every user wants. We can let each user write a program and run them in abstract parallelism by sharing a single computer.

1.4 Multi-tasking

Further generalization leads to *multi-tasking*: solving a problem by decomposition into several concurrent processes. When this is done careful attention must be paid to the *grain* of concurrency: how small are the tasks going to be? The smaller the tasks, the more potential concurrency but the higher the overhead.

Small-grained concurrency requires hardware support to be efficient. A simple example is a computer with separate processors for multiplication and addition. In Figure 1.4, `A*B` and `C+D` can be done in parallel, followed by the second addition. An *array* processor is able to perform a single operation on all elements of an array in parallel (Figure 1.5). While small-grained concurrency can be modeled with the help of the abstractions described in this book, ultimately it is closely connected with specific machine architectures and specific application problems and is better studied in those frameworks. Thus we turn our attention to large-grained concurrency.

```
E := (A*B) + (C+D);
```

Figure 1.4 Parallel function units

```
for I in 1..100 loop
  A(I) := B(I) + C(I)*D(I);
end loop;
```

Figure 1.5 Parallel vector processing

At the other extreme is concurrency on the level of independent programs. Multi-programming operating systems are an obvious example. The Unix operating system provides a convenient mechanism for running programs concurrently. The output of one program may be directly connected to the input of another via a *pipe* (Figure 1.6). The operating system causes an output buffer of one program to be passed as an input buffer of the next program, rather than having the data written to disk and read back in. Not only does this avoid unnecessary disk access, but also it enables programs to be run incrementally making it possible to examine the output as the programs are running.

```
sort | remove_duplicates | format | print
```

Figure 1.6 Pipes in Unix

The typical problem we have in mind for large-grained concurrent programming is one which is divided into a 'few' processes which co-operate to solve the problem, or one where several processes compete for resources. *Real-time embedded* systems are an important class of systems that are constructed using multi-tasking. These are computers that are part of larger systems: aircraft, radar installations, X-ray scanners, power plants. They are required to sample environmental inputs, do calculations and control outputs all within strict requirements on response times that can be as short as several milliseconds. A real-time system is constructed as a set of processes which are scheduled to meet the timing constraints. Constructing real-time systems is particularly challenging because existing formalisms do not adequately cover absolute timing requirements. Nevertheless, the languages and algorithms of concurrent programming are the tools

used to create real-time systems.

Finally, it can be useful to decompose an ordinary sequential program into a multi-tasking program. Multi-tasking can simplify the solution of a problem and it can be used to improve performance on a *multi-processor* – a computer with several processors, or a *multi-computer* – a system with several complete computers connected together.

Consider the outline of a word-processor:

- Read characters and collect as words.
- Collect words to fill a line.
- Hyphenate, if necessary.
- Introduce spaces to justify the line with the right margin.
- Collect enough lines to make a page.
- Print the page.

This is an 'ordinary' program that can be sequentially executed from start to finish. Nevertheless, it is not easy to program sequentially. For example, hyphenation may require a portion of a word to be 'returned' to the stream of words to await the next line. The program will be much easier to understand if it is written as a set of concurrent processes even if it is to be run sequentially.

An example of performance improvement is the Mergesort algorithm (Figure 1.7). As a sequential algorithm, we sort each half of the array and then merge the results. If we have a multi-processor, the two sorts can be done in true parallelism to improve performance. For example, if the input sequence is $(4,2,7,6,1,8,5,0,3,9)$, the two `Sort` procedures can be run in parallel on $(4,2,7,6,1)$ and $(8,5,0,3,9)$ to give $(1,2,4,6,7)$ and $(0,3,5,8,9)$, respectively. `Merge` combines these to give the final result: $(0,1,2,3,4,5,6,7,8,9)$. We need a concurrent programming language to express the fact that the two calls to `Sort` can be done in parallel and to ensure that `Merge` commences only after they have both terminated.

```
procedure Merge_Sort is
  A: array(1..N) of Integer;
  procedure Sort(Low, High: Integer);
  procedure Merge;
begin
  Sort(1,     N/2);
  Sort(N/2+1, N);
  Merge;
end;
```

Figure 1.7 Mergesort

Actually, there is more potential parallelism in this algorithm. If `Sort` produces its output incrementally in ascending order, `Merge` can also be run in parallel. The partial results $(1,2)$ and $(0,3)$ can be merged to obtain a partial answer $(0,1,2)$ without waiting for the rest of the data. To implement this parallelism will require synchronization and communication. Each merge step must be synchronized with the production of a new result from one of the sorts and these results must be communicated from `Sort` to `Merge`. Concurrent programming structures can express the required parallelism and provide programming instructions for synchronization and communication.

1.5 An Outline of the Book

The book is divided into three parts. The first deals with concurrent programming in *common-memory* architectures. These are computers in which processes share access to memory locations or are able to call a centralized operating system to receive a service. Common-memory algorithms are appropriate for concurrent programs that execute by sharing a single computer. They may also be appropriate for multi-processor architectures that have shared memory.

Chapter 2 describes the abstraction that is used (interleaved execution sequences of atomic instructions), defines what it means for a concurrent program to be correct and introduces the methods used to prove correctness. The simplest atomic instruction is a single access to a memory location. Chapter 3 develops algorithms for this model. The chapter is central because it demonstrates in full detail the possible pathological behaviors that a concurrent program can exhibit, as well as verification techniques that can be used to prove correctness.

Chapters 4 and 5 treat the classic concurrent programming primitives, the semaphore and the monitor, that are implemented in terms of an underlying operating system. The ideas are reinforced in Chapter 6 where the problem of the dining philosophers is solved.

Part II deals with *distributed systems* which communicate by exchanging messages. They are intended to model systems where processes execute on physically separated processors.

After an introductory chapter (7) defining the models, three languages are presented in Chapters 8–10: *Ada*, *occam* and *Linda*. These languages are all intended for distributed systems, but their designers chose radically different primitives for expressing concurrency. Comparing the languages gives a broad perspective on the field of distributed computation.

Chapters 11–13 describe three algorithms for distributed systems:

Mutual exclusion The distributed version of the problem studied in Chapter 3.

Termination How can a set of distributed processes agree that a global state exists?

Byzantine Generals An introduction to the subject of fault-tolerance in distributed systems.

Part III discusses the implementation of concurrent programs. Chapter 14 describes the implementation of concurrent programming on a single processor computer. This gives an overview of some of the topics usually taught in a course on 'operating systems'. Chapter 15 presents some examples of the implementation of the three distributed languages: Ada, occam and Linda. Finally, Chapter 16 studies real-time systems where concurrent programming techniques may need modification to ensure correctness under constraints on response time.

The examples are written in the Ada programming language. They use as little of the language as is necessary to demonstrate concurrent programming. In particular, the complex type system is not employed. Appendix A briefly describes the features of the language that are used so that the reader unacquainted with Ada will be able to follow the examples.

Appendix B explains how to convert the fragmentary examples in the text into executable Ada programs. It also describes an implementation of semaphores and monitors in Ada.

Appendix C contains an emulation of the occam and Linda concurrent programming primitives in Ada. Finally, Appendix D describes an implementation of distributed algorithms of Part III.

1.6 Concurrent Programming Problems

The problems in this section can be used as programming exercises. Following each problem is either a reference to the chapter where the problem is discussed, or the outline of one or more solutions. The solutions differ in the amount of concurrency in the program. In a real system, a solution should be chosen that matches the parallelism in the computer and the available synchronization primitives.

1. Mergesort (this chapter).
2. Bounded buffer (Chapter 4).
3. Readers and writers (Chapter 5).
4. Dining philosophers (Chapter 6).
5. Matrix multiplication (Chapter 9, 10).
6. The sum of a set of N numbers.
   - Solution 1: Two processes each compute the sum of $N/2$ numbers from the set. A third process receives the two sums and computes the final answer.
   - Solution 2: Create a binary tree of processes of depth d. This tree has 2^d leaves, each of which computes the sum of $N/2^d$ numbers and passes the partial results to its parent.. An interior process adds the values passed to it by its sons. The output of the root process is the final answer.
   - Solution 3: A set of k processes each has access to the entire set of numbers. A process removes two numbers, adds them and returns the result to the set. When there is only one number in the set, the program terminates and returns that value.
7. (Conway) The program reads 80-character records and writes the data as records of 125 characters. An extra blank is appended after each input record and every pair of asterisks (**) is replaced by an exclamation point (!).
   - Solution: Use three processes, one to read and decompose the input records, one to filter the asterisk pairs and one to compose the output records.

8. (Manna and Pnueli) Compute the binomial coefficient:

$$(n\ k) = n(n-1)\ldots(n-k+1)/1 \times 2 \ldots k$$

   - Solution 1: One process computes the numerator and a second process computes the denominator. A third process does the division.
   - Solution 2: Note (prove) that $i!$ divides $j(j+1)\ldots(j+i-1)$. The numerator process can receive partial results from the denominator process and do the division immediately, keeping the intermediate results from becoming too big. For example, 1×2 divides 10×9, $1 \times 2 \times 3$ divides $10 \times 9 \times 8$, and so on.
9. (Roussel) Given two binary trees with labeled leaves, check if the sequence of labels is the same in each tree. For example, the two trees defined by the expressions $(a,(b,c))$ and $((a,b),c)$ have the same sequence of leaves.
   - Solution: Create two processes to traverse the trees concurrently. They will send the leaf labels in the order encountered to a third process for comparison.
10. (Dijkstra) Let S and T be two disjoint sets of numbers where s and t are the number of elements in the sets. Write a program which modifies the two sets so that S contains the s smallest members of $S \cup T$ and T contains the t largest members of $S \cup T$.
    - Solution 1: Process Ps finds the largest element in S and sends it to Pt which then finds the smallest element in T and sends it to Ps. What is the termination condition?
    - Solution 2: Create $S \cup T$. Let process Ps extract the s smallest elements and Pt extract the t largest elements.
11. (Conway) The game of Life. A set of cells is arranged in a (potentially infinite) rectangular array so that each cell has eight neighbors (horizontally, vertically and diagonally). Each cell is 'alive' or 'dead'. Given an initial finite set of 'alive' cells, compute the configuration obtained after a sequence of generations. The rules for passing from one generation to the next are:
    (a) If a cell is alive and has less than two live neighbors, it dies.
    (b) If it has two or three live neighbors, it continues to live.
    (c) If it has four or more live neighbors, it dies.
    (d) A dead cell with exactly three live neighbors becomes alive.
    - Solution: Each cell is simulated by a process. There are two problems to be solved. Firstly, the computation of the next generation must be synchronized, i.e. the modification of every cell must be based on the status of neighboring cells in the same generation. Secondly, a large data structure of processes and communications channels must be created.

12. (Hoare) Write a disk server that minimizes the amount of seek time done by the arm of a disk drive. A simple server could satisfy requests for data in decreasing order of distance from the current position of the arm. Unfortunately, this could starve a request for data if closer requests arrive too fast.

    - Solution: Maintain two queues of requests: one for requests for data from track numbers less than the current position and one for requests with higher track numbers. Satisfy all requests from one queue before considering requests from the other queue. Make sure that a stream of requests for data from the current track cannot cause starvation.

13. Compute all prime numbers from 2 to n. Note (prove) that if k is not a prime, it is divisible by a prime $p(k) \leq \sqrt{k} + 1$.

    - Solution 1: Allocate a process for each number k from 2 to n. Check if k is divisible by any number less than or equal to $\sqrt{k} + 1$.
    - Solution 2: Allocate processes for each k and have them delete all multiples of k in the set 2 to n.
    - Solution 3 (Sieve of Eratosthenes): Allocate a process to delete all multiples of 2. Whenever a number is discovered to be prime by all existing processes, allocate a new process to delete all multiples of this prime.
    - Solution 4: Divide the set into i blocks of size n/i. Allocate a process to each block. Begin computing the primes in the first block. When a prime k has been computed, we can release all processes containing numbers up to k^2.

1.7 Further Reading

Concurrent programming was originally developed within the context of multiprogramming operating systems and most books on the subject contain chapters on concurrent programming ([Dei85], [PS85]). Interest in parallel algorithms is increasing since the introduction of parallel computers. [Qui87] describes algorithms for these computers.

1.8 Exercises

1. Given partial results of the sort procedures $(0, 3)$ and $(1, 2)$, can the merge procedure immediately create the partial result $(0, 1, 2, 3)$?
2. Describe a concurrent version of Quicksort.

Chapter 2

The Concurrent Programming Abstraction

2.1 Introduction

Scientific descriptions of the world are based on abstractions. One creates an idealized model of some phenomenon and then proceeds to study a higher-level complex system. For example, a human body is described in medical science as a system constructed of organs, bones, nerves, etc. All of these are made up of *cells*, but a physician does not usually need to reason about the properties of individual cells in order to make a diagnosis. Continuing down the scale of abstractions, cells are composed of *molecules* which are composed of *atoms* which are composed of *elementary particles*. As the level of abstraction descends, the abstractions studied become more mathematical since the 'actual' atoms and elementary particles can only be directly perceived under expensive laboratory conditions. It is beyond human ability to make a medical diagnosis based on the quantum theory of atoms, even though the illness is ultimately caused by interactions among the atoms.

In computer science, abstractions are just as important. A programming language is nothing more than an abstraction which is designed to ignore the details of specific machine instructions and architectures. Descending in level of abstraction, we find:

Operating systems An operating system defines a set of systems services, like a file system and operations to access it. The operating system can be implemented on different computers.

Instruction sets Most computer manufacturers design and build families of CPUs which execute the same instruction set as seen by the assembler programmer or compiler writer. The members of a family may be implemented in totally different ways - emulating some instructions in software or using memory for registers.

Integrated circuits Thousands or hundreds of thousands of circuit elements can be packed in one small 'chip'. The abstraction is the description of the behavior of the chip as seen from a few dozen connector pins. Competitors may build a totally different circuit implementing the same abstraction.

Electromagnetics The behavior of individual circuit elements is described by

mathematical models such as Ohm's law and Maxwell's equations dealing with currents, voltages and fields.

Electronics This behavior is explained by the properties of electrons flowing in conductors, semiconductors and insulators.

Quantum theory This theory explains why the electron flow should behave as it does in materials made of different atomic elements.

Again, what is important to understand is that we could never describe the effect of `X:=X+1` directly in terms of the behavior of the electrons within the 'chip' implementing the computer which is executing the statement.

Abstractions are of overwhelming importance in software, though this was not obvious in the early days of programming. Software systems are the most complex systems ever built in terms of the number of different components and the complexity of their interconnection. Two of the most important techniques used in creating software abstractions are encapsulation and concurrency.

Encapsulation acheives abstraction by dividing a software module into a public specification and a hidden implementation. The specification describes the available operations on a data structure or real-world model. The detailed software implementation of the structure or model is written within a separate section that is not accessible outside the module. Thus changes in the internal data representation and algorithm can be made without affecting the programming of the rest of the system. Modern programming languages like Ada directly support encapsulation.

Concurrent programming is an abstraction that is designed to make it possible to reason about the dynamic behavior of programs. This abstraction will be carefully explained in the rest of this chapter.

An objection may be made against abstractions in general, and against concurrent programming in particular, namely that real-world objects are not so neat. If so, the abstraction may be irrelevant for engineering a real system. There are two parts to the answer. The first part is that abstractions are carefully chosen so as to be as close as possible to reality. Otherwise, the abstraction is wrong and a different one must be invented. Secondly, abstractions form the scientific basis upon which the engineer works. In fact the work of an engineer might be described as choosing the best abstraction and implementing it in a way that solves the problem. 'Best' may involve criteria such as cost, performance, reliability, deadlines, etc. It is much easier to modify a known concept to meet some criterion than it is to develop a new concept for each problem. Abstractions are as important to a software engineer as differential equations are to a hardware engineer even though there are no 'ideal' components in either field.

2.2 Interleaving

Concurrent programming abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes.

As discussed in the first chapter, a concurrent program consists of a set of sequential processes executing simultaneously. Each process is an 'ordinary' program with its own code and data and is considered to be executed on a computer which consists of a CPU, memory, and I/O channels. A *distributed* system is a computer system with more than one computer. The computers are connected together and can exchange messages or access each other's memory. If we have one computer for each process, then the correspondence between the abstraction and reality is very close.

However, even if the concurrent program is being executed by sharing the resources of one computer, it is still convenient to consider each process as executing simultaneously. The alternative would be to consider the program as one process which happens to be composed of several modules. The problem with this alternative is the combinatorial explosion of possible program states. For example, in our concurrent Mergesort program (Figure 1.7), let us suppose that each of the three procedures contains ten instructions. We would have to consider $10^3 = 1000$ different possible states of the program. In the case of an operating system, it would not even be possible to predict what computation will be done by each process since the processes ('jobs') are written by programmers working independently.

Thus in our abstraction, each process is considered to be operating on its own processor, executing its own program. We will only have to consider possible interaction in two cases:

Contention Two processes compete for the same resource: computing resources in general, or access to a particular memory cell or channel in particular.

Communication Two processes may need to communicate causing information to be passed from one to the other. Even the mere fact of communication can be important because it allows the processes to *synchronize*: to agree that a certain event has taken place.

So far, we have a set of processes executing simultaneously on a set of processors. These processors can be operating at arbitrary speeds and responding to arbitrary external signals. A further abstraction is to ignore time and to consider only the sequences of instructions as executed by each process. The sequence of instructions depicted in the upper line in Figure 2.1 have different execution times and these vary from one computer to another. We will model them by the sequence in the lower line of the figure where each instruction takes a fixed amount of time. In fact, since the time interval is fixed, we ignore the time scale

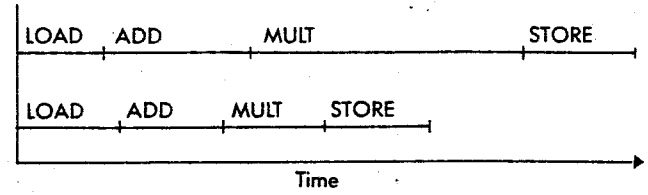

Figure 2.1 Execution sequences

(say, microseconds) and consider the execution of one instruction to be one unit of 'time'.

The justification for this abstraction is that as far as the *correctness* of a program is concerned, the absolute time of execution is not relevant. If we have instructions `End_Print` and `Start_Print` in processes $P1$ and $P2$, respectively, it is important that `End_Print` be executed before and not during or after the execution of `Start_Print` but it should not make a difference if `End_Print` were executed 10 milliseconds or 10 seconds before `Start_Print`.

This abstraction is the hardest one to accept, especially if one has experience with time-critical systems like real-time controllers or operating systems. In these systems, the specification of the program typically contains absolute time requirements: 'update the position of the aircraft on the radar screen within 0.5 seconds'. There are several justifications for the use of execution sequences as the basic model rather than absolute time:

- Execution sequences are appropriate for reasoning about properties other than absolute timing requirements. For most requirements, absolute time would introduce a complexity which is not necessary.
- Systems are always being upgraded with faster components and even faster algorithms. If the correctness of a concurrent program depended on absolute time, every modification to the hardware or software would require that the system be rechecked for correctness. For example, suppose that an operating system was correct under the assumption that characters are being typed in at no more that 10 characters per terminal per second. That is a conservative assumption for a human typist that would be invalidated if a new terminal was installed that did local editing and sent the data on the screen to the computer at 100 characters per second.
- Most time functions, like interrupts from a hardware timer, are actually events that fit into the execution sequence model. Thus, aside from requirements on response time, it is usually possible to deal with time in the model.

In the end, however, the concurrent programming model of execution sequences must be carefully applied or possibly modified or extended when used in systems with absolute time requirements.

The abstraction that ignores absolute speed in a single process is extended so that all the execution sequences of all the processes are *interleaved* into a single execution sequence. Given two instructions $I1$ and $I2$ in two processes $P1$ and $P2$, we can distinguish two cases (Figure 2.2):

1. $end(I1) \leq begin(I2)$
2. $begin(I2) < end(I1)$

The first case is obviously compatible with the concept of a single execution sequence. We forbid the second case by restricting our model so that the effect of two simultaneous instructions is required to be the same as *either* of the two possible sequences obtained by executing the instructions one after the other.

Digital equipment is designed so that any bit of memory or any signal is considered to have one of two binary values. In case of contention, like two

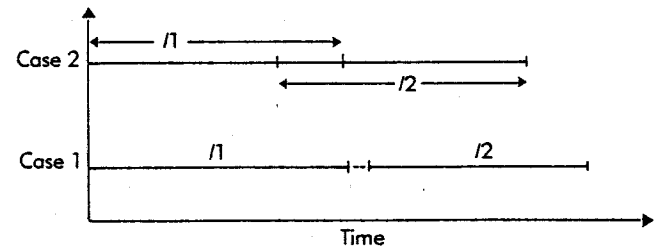

Figure 2.2 Simultaneous execution of instructions

attempts to write to the same memory cell, or two peripherals trying to cause an interrupt, the hardware resolves the situation in a manner compatible with our restriction.

Even in the case of a distributed system, it may be true that two instructions are executed simultaneously on two different computers, but we can arbitrarily assign them an order with no effect on the computation unless there is contention or communication. If the two instructions attempt to access the same hardware, this must be resolved by the hardware. If they send messages on a communications line, we assume an underlying protocol that correctly delivers the messages, though possibly in some arbitrary order.

The model is not as restricted as it might seem on first glance. *Arbitrary* interleavings are allowed. One process could complete hundreds of instructions before any other process executes one instruction. Or we could have a perfect interleaving of one instruction at a time from each process.

A concurrent program is required to be correct under *all* interleavings.

Then, if the computer hardware is changed or if the rate of incoming signals changes, we may have a different interleaving, but the program is still correct.

Furthermore, any concurrent program that depends on external signals is difficult, if not impossible, to repeat exactly. A concurrent program cannot be 'debugged' in the familiar sense of diagnosing a problem, correcting it and rerunning the program to check if the bug still exists. We may just execute a different interleaving in which the bug does not exist. The solution is to develop programming and verification techniques that ensure that a program is correct under all interleavings.

The only constraint to arbitrary interleaving is that *fairness* must be preserved. At every point in the interleaved sequence, *eventually* instructions from each process must be included. It should be intuitively clear that all this is saying is that no processor in a distributed system is infinitely slow, or that no process in a shared system is deferred forever.

A convenient device for reasoning about concurrent programs is to assume that given the instruction sequences of a set of processes, a *scheduler* is in charge of constructing the interleaving. If we are trying to show that a program is incorrect, we take the part of the scheduler and construct a 'bad' interleaving whose behavior

does not meet the problem specification. To prove that a program is correct is more difficult – we must show that no matter how malicious the scheduler is, the behavior of the program is correct.

2.3 Atomic Instructions

The concurrent programming abstraction deals with interleaved sequences of atomic instructions. In applying the abstraction, it is extremely important to define exactly what instructions are being interleaved. For example, suppose that a memory cell represented by an integer variable is being incremented by two processes (Figure 2.3). If the compiler translates the high-level language statement into a single INC instruction, *any* interleaving of the instruction sequences of the two processes will give the same value (Figure 2.4). On the other hand, if all computation is done in registers,[1] the compiled code would look like Figure 2.5. The same figure shows that some interleavings give the wrong answer.

```
N: Integer := 0;

task body P1 is
begin
  N := N + 1;
end P1;

task body P2 is
begin
  N := N + 1;
end P2;
```

Figure 2.3 Two processes incrementing a variable

Process	Instruction	Value of N
(Initially)		0
P1	INC N	1
P2	INC N	2

Process	Instruction	Value of N
(Initially)		0
P2	INC N	1
P1	INC N	2

Figure 2.4 Computer with INC instruction

Thus the correctness of a concurrent program depends on the atomic instructions used by the computer or system on which it runs. In this book, the emphasis

[1] Each process is assumed to have a separate set of registers.

Process	Instruction	N	Reg(1)	Reg(2)
(Initially)		0	-	-
P1	LOAD Reg, N	0	0	-
P2	LOAD Reg, N	0	0	0
P1	ADD Reg, #1	0	1	0
P2	ADD Reg, #1	0	1	1
P1	STORE Reg, N	1	1	1
P2	STORE Reg, N	1	1	1

Figure 2.5 Computer with registers

is not so much on solving a variety of problems as on solving a few typical problems under a wide variety of assumptions about the underlying atomic instructions. The range of possibilities reflects a basic conflict in the design of computer hardware and software. If the instructions are simple, the implementation will be simple and presumably efficient and inexpensive. However, the instructions will be hard to use. If the atomic instructions are more powerful and expressive, it will be easier to write correct programs, but the implementation will be more difficult.

The example of Figure 2.3 illustrates that it is harder to write a correct program for a computer that must do arithmetic in a register than for one which can compute directly to memory.

The atomic instructions used in Part I assume the existence of common memory accessible to all processes. Common memory can be used in two ways which differ only in what is accessed by the processes:

- Global data which may be read and written by more than one process.
- Object code of operating system routines which may be called more than one process.

In Part II we assume only that each process can send a message to another process.

Common memory instructions are easy and efficient to implement on a single computer being shared by the various processes. Message passing is needed in distributed systems. In many cases, we will show how one atomic instruction can be emulated by an algorithm written using another instruction. The emulations may not be efficient or simple but they give a feeling for the tradeoffs between expressiveness and simplicity.

The first atomic instruction studied is Load-Store to common memory. This is the model shown in Figure 2.5 where *every* access to a global variable is a single instruction and interleaving is permitted between any two accesses. On the other hand, an attempt to simultaneously access a variable is resolved as an interleaving, not as a garbage value. It is possible to weaken this assumption even further (and on one occasion we shall do so), but in most practical applications this is as low-level as necessary.

Load-Store is not very powerful and solutions to concurrent programming problems that use this primitive are not efficient. Most concurrent programs are written on machines that support complex instructions (like INC) or high-level

primitives implemented on an underlying operating system.

2.4 Correctness

For sequential programs, the concept of *correctness* is so familiar that the formal definition is often neglected. For concurrent programs, it is imperative to formalize correctness, both because the concepts are very different and often unintuitive, and because formal verification of correctness is often crucial.

There are two definitions of correctness for programs that are supposed to terminate. Let $P(\vec{x})$ be a property of the input variables $\vec{x}$ and $Q(\vec{x},\vec{y})$ be a property of the input variables $\vec{x}$ and output variables $\vec{y}$. Then for any values $\vec{a}$ of the input variables, we define correctness as follows:

Partial correctness If $P(\vec{a})$ is true *and* the program terminates when started with $\vec{a}$ as the values of the input variables $\vec{x}$, then $Q(\vec{a},\vec{b})$ is true where $\vec{b}$ are the values of the output variables upon termination.

$$(P(\vec{a}) \wedge terminates(Prog(\vec{a},\vec{b}))) \supset Q(\vec{a},\vec{b}) \tag{2.1}$$

Total correctness As in partial correctness, but we require the program to terminate.

$$P(\vec{a}) \supset (terminates(Prog(\vec{a},\vec{b})) \wedge Q(\vec{a},\vec{b})) \tag{2.2}$$

To give a trivial example, if `SQR(x,y)` is a program that computes the square root of `x` and stores it into the variable `y`, the two specifications of correctness give:

$$(a \geq 0 \wedge terminates(SQR(a,b))) \supset b = \sqrt{a} \tag{2.3}$$

$$a \geq 0 \supset (terminates(SQR(a,b)) \wedge b = \sqrt{a}) \tag{2.4}$$

Note that the behavior of the program on negative numbers is irrelevant. If we were concerned about the behavior of the program when negative numbers are input, the specification of correctness would have to be changed accordingly. Note also that a program that always goes into an infinite loop is partially correct for this specification (in fact, it is partially correct for *all* specifications).

Partial correctness and total correctness are appropriate for concurrent programs which terminate just as for sequential programs. But when specifying non-terminating programs such as operating systems or real-time controllers, we need other definitions. If one of these systems terminates, we consider it a bug!

Correctness of concurrent programs is defined in terms of properties of execution sequences. Such a property is (implicitly) prefixed by: 'for all possible execution sequences'. This was discussed earlier in the chapter as a means of modeling different relative speeds of the processors as well as the non-repeatability of concurrent programs.

There are two types of correctness properties:

Safety properties The property must *always* be true.

Liveness properties The property must *eventually* be true ('now' is included in 'eventually').

The most common safety property is *mutual exclusion*: two processes may not interleave certain (sub-)sequences of instructions. Instead, one sequence must be completed before the other commences. The order in which the sequences are executed is *not* important. This models many problems in operating systems such as access to resources like printers and access to system-wide data and services. It must *always* be true that at most one process access a printer at any one instant. It is inconceivable that an operating system could be considered correct if it ever assigned a printer to two processes, even if only momentarily and even if only under unlikely circumstances.

The other important safety property is absence of *deadlock*. A non-terminating system must always be able to proceed doing useful work. A system which cannot respond to any signal or request is deadlocked and again, this must never happen. A system may be quiescent if there is nothing to do, but in that case, it is usually actively executing a background idle-process or self-test and it will be able to respond instantly to an interrupt. A deadlocked system will not respond, or as the slang goes, the system is 'hung'.

Like partial correctness which can be satisfied by any non-terminating program, a program which does nothing will satisfy most safety properties. A program which never assigns a printer to any process will trivially not assign one to two processes. In order for something to actually happen, liveness properties must be specified. For example, if a process posts a request to print, *eventually* it will be assigned a printer. The technical term is *absence of (individual) starvation*. This is a weak specification because a program that assigned a printer after a thousand years would still be correct, but we have already justified ignoring absolute time. If the system is not malicious, 'eventually' will be relatively soon unless there is contention for the printer.

A particular type of liveness property is called a *fairness property*. If there is contention, we will often want to specify how the contention must be resolved. Four possible specifications of fairness are:

Weak fairness If a process continuously makes a request, eventually it will be granted.

Strong fairness If a process makes a request infinitely often, eventually it will be granted.

Linear waiting If a process makes a request, it will be granted before any other process is granted the request more than once.

FIFO (first-in first-out) If a process makes a request, it will be granted before that of any process making a later request.

The difference between strong and weak fairness is shown in Figure 2.6. A process requests a service by setting a flag to 1. The upper line shows process `P1` periodically issuing a request and then canceling the request whereas process `P2` depicted in the lower line issues a request and holds it outstanding for an indefinite period. The process granting the requests checks the flag periodically at

times $t0, t1, t2, \ldots$. As shown, this system is weakly-fair because it will grant P2's request but not strongly-fair since it will not grant P1's request. Even though P1 makes its request infinitely often, the server may check the outstanding requests exactly when P1 is not requesting. A strongly-fair system *would* grant P1's request.

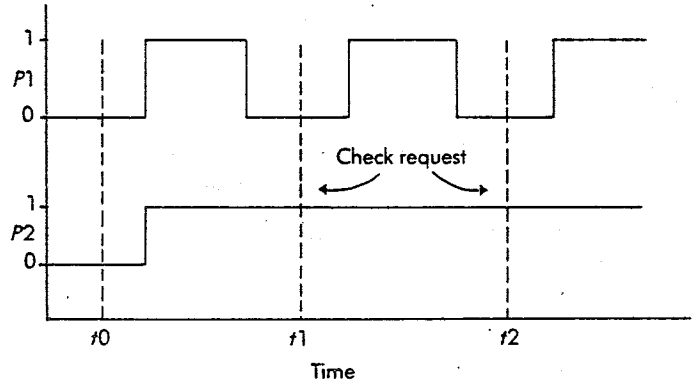

Figure 2.6 Strong and weak fairness

These two concepts of fairness are not very practical because they depend on 'eventually' and 'infinitely often'. Linear waiting is a practical concept because it says that once a process has issued a request, it may be overtaken by every other process, but only once. If we know what the longest permitted print job is, we can calculate a bound on the length of time a process must wait to be assigned a printer.

FIFO is the strongest specification of fairness. It is simple to implement on centralized systems. On a distributed system, it is not clear what 'later' means, so weaker definitions of fairness are important.

2.5 Inductive Proofs of Correctness

A safety property of a concurrent program holds if it is true at every state of every execution sequence. A non-terminating program is potentially infinite and there are an infinite number of execution sequences since at each step the scheduler could choose an instruction from any process to create the interleaving. How can one prove properties of infinite objects?

The answer is proof by induction. Let us recall how this works in mathematics before describing the technique for programs. If a property $P(n)$ is to hold for all natural numbers n (i.e. for all $n \geq 0$), we prove it in two steps:

Basis Prove $P(0)$.

Inductive step Assume that $P(n)$ is true (the *inductive hypothesis*) and prove $P(n+1)$.

Intuitively, if $P(0)$ is true, by the inductive step applied once, $P(1)$ is true. Now assuming this as the inductive hypothesis, we prove $P(2)$. There is no bound on the value of n for which we can prove $P(n)$, so $P(n)$ is true for all n.

The inductive proof technique is valid for any sequence of values that can be put into one-to-one correspondence with the natural numbers: $i_0, i_1, \ldots$. In particular, we can use it to prove that a property holds *always* during an execution sequence:

Basis Prove the property for the initial state of the program.

Inductive step Assume that the property holds after the n^{th} step of the execution sequence (the inductive hypothesis) and prove that it holds after the $(n+1)^{\text{st}}$ step.

But we do not know what the execution sequence is until the scheduler actually chooses a process whose instruction will be executed next. So we are required to prove that if the inductive hypothesis holds in any state, it will continue to hold in *each* state that can be reached by executing a statement of any process. Since any execution sequence can be constructed as an interleaving of the individual execution sequences, the property holds since the induction holds for each choice made by the scheduler. A property which is true for each state of every possible execution sequence is called an *invariant*.

Figure 2.7 is a trivial program designed to demonstrate the technique. We want to prove that $P \equiv X = 0 \vee X = 1$ is an invariant. It is impossible to prove this directly. Obviously, it is true initially. Assuming that P is true in any state, we have to prove that P remains true after any instruction of the program is executed. Suppose we are at a state where process $P1$ is about to execute `X := X+1`. Using our knowledge of the semantics of the assignment statement, we can conclude that $X = 1 \vee X = 2$, which is not what we want.

As usual with inductive proofs, the solution is to prove a stronger claim than the required property. The invariant will include formulas $at(l)$ expressing the value of the instruction pointer of a process. $at(l)$ means that the location l is the next instruction to be executed by that process.

Theorem 2.5.1 $Q \equiv (at(a1) \supset X = 0) \wedge (at(b1) \supset X = 1)$ *is an invariant.*

Proof: Initially, process $P1$ is $at(a1)$ and the value of X is 0, so Q is true.

Assume that Q is true in any state. There are four possible transitions in the program. For each one we will prove that Q remains true following the transition.

1. $a1 \rightarrow b1$: By the inductive hypothesis, when $P1$ is $at(a1)$, then $X = 0$. Following the transition, $at(b1)$ is true, but so is $X = 1$ by the semantics of the assignment statement.

2. $b1 \rightarrow a1$: By the inductive hypothesis, $X = 1$. Following the transition, $at(a1)$ and $X = 0$ are true.

3. $a2 \rightarrow b2$: A transition in $P2$ does not affect the location counter of $P1$ nor the value of X. Since the truth of Q depends only on these values, the inductive hypothesis trivially implies that Q is still true.

4. $b2 \rightarrow a2$: As in (3). □

```
task body P1 is
  X: Integer := 0;
begin
  loop
a1:  X := X + 1;
b1:  X := X - 1;
  end loop;
end P1;

task body P2 is
  Y: Integer := 0;
begin
  loop
a2:  Y := Y + 1;
b2:  Y := Y - 1;
  end loop;
end P2;
```

Figure 2.7 Inductive proofs of correctness

Since $at(a1) \vee at(b1)$ is true, theorem 2.5.1 implies that P is invariant.

Note that (1) and (2) require reasoning only on process $P1$, and (3) and (4) only on $P2$. This shows how concurrent programming solves combinatorial explosion since we need to prove the invariant on a number of cases equal to the sum and not the product of the number of instructions in each process. (3) and (4) are called proof of *non-interference* since they show that $P2$ cannot interfere with $P1$.

In many cases, proofs can be simplified using elementary propositional logic. An implication $P \supset Q$ is true if and only if the *antecedent* P is false or the *consequent* Q is true. It will remain true, unless P and Q are both false and P 'suddenly' becomes true by executing a statement, or P and Q are both true and Q 'suddenly' becomes false. Many inductive formulas will take the form of an implication:

$$at(location) \supset variable = value$$

If it is assumed true then it could only become false on a transition to *location* when *variable* $\neq$ *value* or a transition by another process that causes the consequent to become false while this process is in *location*.

2.6 Liveness proofs

To prove liveness properties is more complicated in that the formalism is more complicated. On the other hand, one need only prove that a certain state must eventually occur on any execution sequence. The arguments are based on the assumed fairness of the underlying scheduler and the eventuality properties of the various program constructs.

For example, we prove that if process $P1$ in Figure 2.7 is $at(a1)$ then eventually it will be at $at(b1)$. Eventually, process $P1$ must be scheduled and we assume that an assignment statement always terminates. Thus eventually $P1$ will be $at(b1)$.

In discussing liveness we will use the following notation:

- $\Box p$ means p is *always* true.
- $\Diamond p$ means p is *eventually* true.

These are operators of a formal logic called *temporal logic* which facilitates reasoning on propositions that change with time. In particular, it can be used to reason about programs which change with time. Thus $\Box p$ means that at every subsequent point in the execution sequence, p is true. This is what we have called an invariant. $\Diamond p$ means that at some future moment in the execution sequence, p is true (where future includes 'now'). The formula from the previous paragraph can be written: $at(a1) \supset \Diamond at(b1)$. A useful formula is $\Diamond\Box p$ which means that at some future moment, p will be true and will stay that way forever.

A formal development of temporal logic is beyond the scope of this book. In the following proofs in temporal logic, use common-sense reasoning as if these operators were just convenient abbreviations for natural language. For example, from $\Diamond\Box p$ and $\Diamond\Box q$, we can deduce $\Diamond\Box(p \wedge q)$. As shown in Figure 2.8, eventually one formula, say p, must be held true and then eventually q must be held true while p is (obviously) still held true. Thus from that moment on, both p and q will be true.

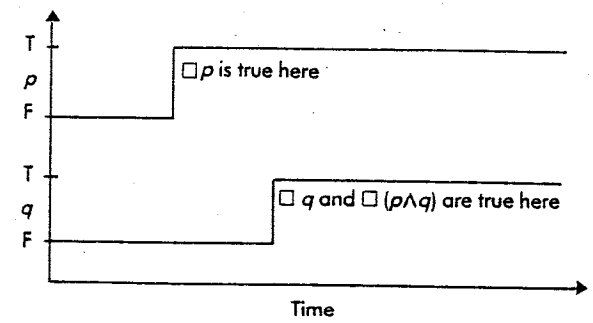

Figure 2.8 Example in temporal logic

Liveness proofs are difficult when there are conditional statements: `if B then S1 else S2`. We can assume that eventually we will execute either statement `S1`

or statement S2 but we cannot know which without more information on the boolean expression B. Even if B is eventually true ($\Diamond B$), it might be true only momentarily and the scheduler may execute the conditional statement before or after this state. To conclude that eventually S1 is executed ($\Diamond at(S1)$), we must first prove an invariant showing that the value of B will be held true indefinitely or that eventually such an invariant becomes true ($\Diamond\Box B$).

2.7 Further Reading

The interleaving model is standard in the literature on concurrent programming. For a different view based on relativity see [Lam86]. [Gri81] is a comprehensive text on verification of sequential programs. Inductive proofs of concurrent programs are from [OG76], and the temporal logic of concurrent programs is from [MP81]. [Fra86] is a research monograph on the difficult subject of fairness.

2.8 Exercises

1. Enumerate all the possible interleavings of instructions in the program in Figure 2.3 under the register model of computation. Note which interleavings give the correct answer.
2. Estimate the number of possible interleavings of concurrent Mergesort as a function of the size of the array.
3. Estimate the number of possible interleavings of concurrent matrix multiplication assuming that each element of the result matrix is computed by a separate process.
4. Draw a diagram similar to Figure 2.8 for: $\Box\Diamond\Box p$ and $\Diamond\Box\Diamond p$. What relations exist between those formulas and $\Box\Diamond p$ and $\Diamond\Box p$?
5. Which of the following are true? In each case draw a diagram justifying your answer.
   (a) $\Box(p \wedge q) \supset (\Box p \wedge \Box q)$
   (b) $(\Box p \wedge \Box q) \supset \Box(p \wedge q)$
   (c) $\Diamond(p \wedge q) \supset (\Diamond p \wedge \Diamond q)$
   (d) $(\Diamond p \wedge \Diamond q) \supset \Diamond(p \wedge q)$
   (e) $\Box(p \vee q) \supset (\Box p \vee \Box q)$
   (f) $(\Box p \vee \Box q) \supset \Box(p \vee q)$
   (g) $\Diamond(p \vee q) \supset (\Diamond p \vee \Diamond q)$
   (h) $(\Diamond p \vee \Diamond q) \supset \Diamond(p \vee q)$

Chapter 3

The Mutual Exclusion Problem

3.1 Introduction

This chapter presents solutions to the mutual exclusion problem. First we will solve the problem for two processes using Load-Store to common memory as the only atomic instructions. Solutions will then be given for N processes using Load-Store and then using more complex atomic instructions.

One solution to the mutual exclusion problem for two processes is called Dekker's algorithm. This algorithm will be developed in a step-by-step sequence of incorrect algorithms. Each of these incorrect algorithms will demonstrate some pathological behavior that is typical of concurrent algorithms. Formal proofs of properties of the algorithms will be emphasized.

The mutual exclusion problem for N processes:

1. N processes are executing in a infinite loop a sequence of instructions which can be divided into to subsequences: the critical section and the non-critical section. The program must satisfy the *mutual exclusion property*: instructions from the critical sections of two or more processes must not be interleaved.
2. The solution will be described by inserting into the loop additional instructions that are to be executed by a process wishing to enter and leave its critical section – the *pre-protocol* and *post-protocol*, respectively (Figure 3.1). These protocols may require additional variables.

```
loop
  Non_Critical_Section;
  Pre_Protocol;
  Critical_Section;
  Post_Protocol;
end loop;
```

Figure 3.1 Form of mutual exclusion solution

3. A process may halt in its non-critical section. It may not halt during execution of its protocols or critical section. If one process halts in its non-critical section.

it must not interfere with other processes.

4. The program must not deadlock. If *some* processes are trying to enter their critical section then *one* of them must eventually succeed. There may be local progress within the protocols as the processes set and check the protocol variables, but if no process ever succeeds in making the transition from pre-protocol to critical section, we say that the program is deadlocked.
5. There must be no starvation of one of the processes. If a process indicates its intention to enter the critical section by commencing the execution of the pre-protocol, *eventually* it will succeed. This is a very strong requirement in that it must be shown that *no* possible execution sequence of the program, no matter how improbable, can cause starvation.
6. In the absence of contention for the critical section, a single process wishing to enter its critical section will succeed. Not only will it succeed, but a good solution to the mutual exclusion problem will have minimal overhead in this case.

The solution may assume that load and store to common memory are atomic. This will be indicated by declaring variables that are global to both processes and allowing either process to store the variable by assignment `C1:=1`, or to load the value of the variable. Syntactically, a load will be indicated by having the variable appear in an expression in the right-hand side of an assignment statement `C2:=C1` or in an expression used in a conditional statement `if C1=1 then`. In the event of a simultaneous load and store, the effect is to interleave the two instructions in an arbitrary order.

3.2 First Attempt

In a first attempt at solving the mutual exclusion problem for two processes (Figure 3.2), we define a single global variable Turn which can take the two values 1 and 2. Initially, Turn has an arbitrary value, say 1. The variable indicates whose 'turn' it is to enter the critical section. A process wishing to enter the critical section will execute a pre-protocol consisting of a statement that loops until the value of Turn indicates that its turn has arrived. Upon exiting the critical section, the process sets the value of Turn to the number of the other process.

Notation: Turn refers to the variable in the algorithm. $Turn$ is used to denote the value of that variable in a formula like $Turn = 1$. A similar distinction is made for other variables.

Theorem 3.2.1 *This solution satisfies the mutual exclusion requirement.*

Proof: A formal inductive proof is left as an exercise that may be done after studying the technique in the proof of the third attempt. Here we give an informal proof.

Suppose, to the contrary, that at some point in the execution sequence both processes are in their critical sections. Without loss of generality, P1 entered its


```
  Turn: Integer range 1..2 := 1;

task body P1 is
begin
  loop
    Non_Critical_Section_1;
    loop exit when Turn = 1; end loop;
    Critical_Section_1;
    Turn := 2;
  end loop;
end P1;

task body P2 is
begin
  loop
    Non_Critical_Section_2;
    loop exit when Turn = 2; end loop;
    Critical_Section_2;
    Turn := 1;
  end loop;
end P2;
```

Figure 3.2 First attempt

critical section before P2. That is, P1 entered its critical section at time $t1$, P2 entered its critical section at time $t2$, $t1 < t2$, and during this time interval, P1 remained in its critical section. Now at time $t1$, $Turn = 1$ so that P1 was able to exit its loop statement. Similarly, at time $t2$, $Turn = 2$. But during the interval from $t1$ to $t2$, P1 remained in its critical section and did not execute the post-protocol, which is the only statement that can assign the value 2 to Turn. Thus the value of Turn at $t2$ must still be 1, contradicting our previous statement. □

Let us examine this proof to see where the assumptions of the concurrent programming model have been used. The assumption that instructions are interleaved means that the processes cannot enter their critical sections simultaneously. If they try to enter simultaneously, they will be interleaved in some arbitrary order. We must prove that the requirement is satisfied under either order. However, given the symmetry of the solution (both processes execute identical programs except for the process number), it is sufficient to prove the theorem for one of the two possibilities.

The assumption that load and store are atomic instructions allows us to claim that the value of Turn must be either 1 or 2 and that it can only change when explicitly assigned.

Theorem 3.2.2 *The solution cannot deadlock.*

Proof: For the program to deadlock, both processes must execute the test on Turn in the loop infinitely often and fail. Then from P1, we can deduce that

$Turn = 2$ and from P2, $Turn = 1$, which is impossible. □

Theorem 3.2.3 *There is no starvation.*

Proof: For starvation to exist, one process, say P1, must enter its critical section infinitely often, while the other process executes its pre-protocol forever without succeeding in entering its critical section. But if P1 executes its critical section even once, it will set Turn to 2 in its post-protocol. Then the next time that P2 is allowed to execute, it will succeed in entering its critical section. □

Theorem 3.2.4 *The solution can fail in the absence of contention.*

Proof: Suppose that P2 halts in the non-critical section. Then the value of Turn will never again be changed from 2 to 1. So process P1 may enter its critical section at most one more time. Once P1 has set Turn to 2, it will never again be able to succeed in the test in its pre-protocol. □

Theorem 3.2.4 shows that we have not solved the mutual exclusion problem because one of the required properties does not hold. In fact, even if both processes can be guaranteed not to halt, we must reject this solution. If P1 needs to enter its critical section several times per second, while P2 is content to enter once per hour, this solution would not solve the problem. The processes are coerced into working at the same rate.

The explicit passing of the right to execute is a well-known programming technique called *coroutines*. This technique is useful when a single problem is to be solved by several modules working together. We are interested in systems that involve several loosely-coupled processes working on independent problems simultaneously or different aspects of a single problem. We explicitly wish to refrain from making assumptions on the relative speed of the various processes. For this, coroutines are not sufficient and the full power of concurrent programming techniques is needed.

3.3 Second Attempt

The previous solution is incorrect because both processes are setting and testing a single global variable. Thus if one process dies, the other is blocked. Let us try to remedy this situation by providing each process with its own variable (Figure 3.3). The interpretation that should be placed on the variable is: $Ci = 0$ if Pi wishes to enter its critical section. Ci will be set by Pi before it enters its critical section and reset upon completion of its critical section. It will be tested by the other process which will loop until the critical section may be safely entered. If a process is halted in its non-critical section, the value of its variable will remain at 1 and the other process will always succeed in immediately entering its critical section.

Unfortunately, the program does not even satisfy the mutual exclusion requirement.


```
C1, C2: Integer range 0..1 := 1;

task body P1 is
begin
  loop
    Non_Critical_Section_1;
    loop exit when C2 = 1; end loop;
    C1 := 0;
    Critical_Section_1;
    C1 := 1;
  end loop;
end P1;

task body P2 is
begin
  loop
    Non_Critical_Section_2;
    loop exit when C1 = 1; end loop;
    C2 := 0;
    Critical_Section_2;
    C2 := 1;
  end loop;
end P2;
```

Figure 3.3 Second attempt

Theorem 3.3.1 P1 *and* P2 *can be in their critical sections simultaneously.*

Proof: Consider the following interleaving of instructions of the two processes beginning with the initial state

1. P1 checks C2 and finds $C2 = 1$.
2. P2 checks C1 and finds $C1 = 1$.
3. P1 sets C1 to 0.
4. P2 sets C2 to 0.
5. P1 enters its critical section.
6. P2 enters its critical section.

This violates the mutual exclusion requirement. □

It is extremely important to note the difference between this proof and the proof that the first attempt does satisfy mutual exclusion. Since this property is a safety property, it must be satisfied for *all* possible execution sequences. Thus we need a mathematical argument to prove it because the number of execution sequences is infinite. To show that the property is false, however, it is sufficient to describe one execution sequence for which it fails.

3.4 Third Attempt

In the second attempt, we introduced the variables Ci which are intended to indicate when process Pi is in its critical section. However, once a process has successfully completed its loop statement, it cannot be prevented from entering its critical section. Thus the state of a computation reached after the loop and before the assignment to Ci is effectively part of the critical section, yet Ci still does not indicate this fact. The third attempt (Figure 3.4) recognizes that the loop should be considered part of the critical section by moving the assignment to Ci before the loop.

```
  C1, C2: Integer range 0..1 := 1;

task body P1 is
begin
  loop
a1:  Non_Critical_Section_1;
b1:  C1 := 0;
c1:  loop exit when C2 = 1; end loop;
d1:  Critical_Section_1;
e1:  C1 := 1;
  end loop;
end P1;

task body P2 is
begin
  loop
a2:  Non_Critical_Section_2;
b2:  C2 := 0;
c2:  loop exit when C1 = 1; end loop;
d2:  Critical_Section_2;
e2:  C2 := 1;
  end loop;
end P2;
```

Figure 3.4 Third attempt

Theorem 3.4.1 *The solution satisfies the mutual exclusion property.*

Proof: The proof will be by induction on the execution sequence. We claim that the following formulas are invariant.

$$C1 = 0 \equiv at(c1) \vee at(d1) \vee at(e1) \tag{3.1}$$
$$C2 = 0 \equiv at(c2) \vee at(d2) \vee at(e2) \tag{3.2}$$
$$\neg(at(d1) \wedge at(d2)) \tag{3.3}$$

Proof of 3.1: Formula 3.1 is initially true because the initial value of C1 is 1 and initially, $at(a1)$ is true, hence both sides of the equivalence are false.

Assume by induction that the formula is true and consider every possible transition that the program can make.

1. $a1 \rightarrow b1$: This transition does not affect the truth of any formula in the invariant. Hence, if it was true before executing this instruction, it is still true.
2. $b1 \rightarrow c1$: This makes the left-hand side of the equivalence true by assigning 0 to C1. It also makes the disjunction on the right-hand side true, because now $at(c1)$ is true.
3. $c1 \rightarrow c1$: This transition means that we execute the test in the loop and fail to exit. This does not affect the truth of left-hand side because C1 is not assigned. The right-hand side remains true.
4. $d1 \rightarrow e1$: As in the previous transition.
5. $e1 \rightarrow a1$: This makes the left-hand side false by assignment and the right-hand side false by the transition out of e1.
6. Any transition in P2. Obviously, C1 is not assigned and the location counter of P1 is not affected by a transition in P2.

Thus we have shown that formula 3.1 is initially true and that following *any* transition of the program (in either process) it remains true. Hence, the formula is an invariant of the computation.

The above proof is done in full detail so that the concept of inductive proofs can be understood. In practice, a more concise argument would suffice:

Proof of 3.1: The formula is true initially. The only transitions that could affect the truth of the formula are the transitions: $b1 \rightarrow c1$ and $e1 \rightarrow a1$. Now show that the two assignments to C1 also cause the location counter to change to preserve the truth of the formula. □

The invariance of 3.2 follows by a symmetric proof. It remains to prove the invariance of 3.3.

Proof of 3.3: The formula is trivially true initially because the initial locations are $a1$ and $a2$. The only transitions that could possibly falsify the formula are $c2 \rightarrow d2$ while $at(d1)$ is true or symmetrically the transition in P2. In words, P1 is in its critical section and then P2 enters.

By invariant 3.1, $at(d1) \supset C1 = 0$. Then it is impossible for the transition $c2 \rightarrow d2$ to occur because the condition in the loop requires $C1 = 1$. By a symmetric argument, we can prove that the transition in P2 cannot occur.

We have shown that the formula is initially true, and that the only transitions that can falsify the formula can never occur. Hence those that do occur preserve the truth of the formula. □

Formula 3.3 is exactly the statement of the mutual exclusion property so the theorem has been proved. □

Unfortunately, the program can deadlock after just a few instructions if the interleaving executes one instruction from each process alternately.

1. P1 assigns 0 to C1.
2. P2 assigns 0 to C2.
3. P1 tests C2 and remains in the loop.
4. P2 tests C1 and remains in the loop.

Now the program will interleave tests all of which fail. Thus we have a situation where a set of processes (both of them) wish to enter the critical section, but no process will ever succeed.

3.5 Fourth Attempt

In the third attempt, when a Pi sets the variable Ci to 0, it not only indicates its intention to enter its critical section, but also *insists* on its right to do so. This can cause deadlock if both processes simultaneously insist on entering their critical sections.

The fourth attempt (Figure 3.5) attempts to remedy the problem by requiring a process to give up its intention to enter its critical section if it discovers that a state of contention with the other process exists. The sequence of assignments to the same variable C1:=1; C1:=0 is meaningful in a concurrent program, unlike a sequential program. Since we allow arbitrary interleaving of instructions from the two processes, P2 may be allowed to execute an arbitrary number of instructions between the two assignments to C1. In this case, when P1 relinquishes the attempt to enter the critical section by resetting C1 to 1, P2 may now execute the loop once more and succeed in entering the critical section.

This solution satisfies the mutual exclusion property for exactly the same reasons that the third attempt did. The transition to the critical section is made only if the variable in the other process is 1 and simple invariants describe exactly when this occurs. However, this solution suffers from two defects: individual starvation is possible, as well as a form of deadlock known as *livelock*. Recall that deadlock means that a set of processes wishes to enter their critical sections, but no process can succeed. In a deadlocked computation, there is *no* possible execution sequence which succeeds. In a livelocked computation, there are successful computations, but it is also possible to describe one or more execution sequences in which no process ever enters its critical section.

Theorem 3.5.1 *A process can be starved.*

Proof: Remember that arbitrary interleavings are possible. Between the two assignments to the variable C2, P1 can complete a full cycle of its outer loop and make another attempt to enter its critical section just as P2 makes another attempt.

1. P1 sets C1 to 0.
2. P2 sets C2 to 0.
3. P2 checks C1 and then resets C2 to 1.
4. P1 completes a full cycle:


```
  C1, C2: Integer range 0..1 := 1;

task body P1 is
begin
  loop
    Non_Critical_Section_1;
    C1 := 0;
    loop
      exit when C2 = 1;
      C1 := 1;
      C1 := 0;
    end loop;
    Critical_Section_1;
    C1 := 1;
  end loop;
end P1;

task body P2 is
begin
  loop
    Non_Critical_Section_2;
    C2 := 0;
    loop
      exit when C1 = 1;
      C2 := 1;
      C2 := 0;
    end loop;
    Critical_Section_2;
    C2 := 1;
  end loop;
end P2;
```

Figure 3.5 Fourth attempt

- Checks C2.
- Enters critical section.
- Resets C1.
- Executes non-critical section.
- Sets C1 to 0.

5. P2 sets C2 to 0.

We have now returned to the state described in line 3. The same steps can be repeated indefinitely, so it is possible to describe an execution sequence in which P1 enters its critical section infinitely often (so obviously this does not describe a deadlocked computation), but P2 remains indefinitely in its pre-protocol. □

Theorem 3.5.2 *The solution can livelock.*

Proof: Consider an execution sequence which perfectly alternates instructions of

the two processes:

1. P1 sets C1 to 0.
2. P2 sets C2 to 0.
3. P1 checks C2 and remains in the loop.
4. P2 checks C1 and remains in the loop.
5. P1 resets C1 to 1.
6. P2 resets C2 to 1.
7. P1 sets C1 to 0.
8. P2 sets C2 to 0.
9. P1 checks C2 and remains in the loop.
10. P2 checks C1 and remains in the loop.

This execution sequence can be continued indefinitely. Thus we have the situation that defines deadlock: two processes wishing to enter their critical section, but none succeeding. However, the slightest deviation from the sequence described will allow one of the processes to enter its critical section. Thus we classify the problem as livelock rather than deadlock. □

It should be clear that the mutual exclusion property and the absence of deadlock are critical correctness requirements that must always be met in a real system. Livelock and starvation may be acceptable if the amount of contention is low so that the probability of these problems is negligible. A system designer must balance the probability and possible effects of these problems against the complexity of a more sophisticated algorithm which does not suffer from them.

3.6 Dekker's Algorithm

Dekker's algorithm (Figure 3.6) is a combination of the first and fourth attempted solutions. Recall that in the first solution we explicitly passed the right to enter the critical section between the processes. Unfortunately, this caused the processes to be too closely coupled and prevented correct behavior in the absence of contention. In the fourth solution, each process had its own variable which prevented problems in the absence of contention, but in the presence of contention neither process had the right to insist on entering its critical section.

Dekker's algorithm is like the fourth attempted solution, except that the right to *insist* is explicitly passed between the processes. The individual variables in each process will ensure mutual exclusion, but upon detecting contention, a process, say P1, will consult an additional global variable Turn to see if it is its turn to insist upon entering its critical section. If not, it will reset C1 and defer to P2, waiting on Turn. When the P2 completes its critical section, it will change Turn to 1, freeing P1. Even if P2 immediately made other requests to enter the critical section, it will be blocked by Turn once P1 re-issued its request.

Dekker's algorithm is correct: it satisfies the mutual exclusion property, it does not deadlock, neither process can be starved and in the absence of contention a process can enter its critical section immediately. The proofs are similar to those


```
  C1, C2: Integer range 0..1 := 1;
  Turn:   Integer range 1..2 := 1;
task body P1 is
begin
  loop
    Non_Critical_Section_1;
    C1 := 0;
    loop                                ---------------------
i1:   exit when C2 = 1;                 -- insisting loop
i2:   if Turn = 2 then                  -- when Turn /= 2
        C1 := 1;
        loop exit when Turn = 1; end loop;
        C1 := 0;
      end if;
    end loop;                           ---------------------
    Critical_Section_1;
    C1 := 1;
    Turn := 2;
  end loop;
end P1;

task body P2 is
begin
  loop
    Non_Critical_Section_2;
    C2 := 0;
    loop                                ---------------------
      exit when C1 = 1;                 -- insisting loop
      if Turn = 1 then                  -- when Turn /= 1
        C2 := 1;
        loop exit when Turn = 2; end loop;
        C2 := 0;
      end if;
    end loop;                           ---------------------
    Critical_Section_2;
    C2 := 1;
    Turn = 1;
  end loop;
end P2;
```

Figure 3.6 Dekker's algorithm

of the previous attempts except for the proof of absence of starvation which we now proceed to do in some detail.

The statement we want to prove is that if P1 begins to execute its pre-protocol then *eventually* it will enter its critical section. A symmetric proof will show absence of starvation for P2. Unlike mutual exclusion which is a property that is *expressed* in terms of the location counters of both processes ($\neg at(d1 \wedge d2)$), absence of starvation can be expressed in terms of the location counters of P1 only. We are going to prove the theorem without using formulas that involve location counters of both processes.

We first prove two 'commitments'. These are theorems on the values of the global variables. They state that under certain assumptions on the behavior of the global variables, we can conclude that eventually something is true of another global variable. The theorems will be proved using only arguments on the structure of P2. Once we have proved that P2 is 'committed' to behave in some manner, we will prove that P1 cannot be starved. The advantage of this type of proof is that it is more elementary, since we have only to inspect the code of one process at a time. In addition, it would presumably be possible to replace the code of P2 by any other program that promised to behave as required by the commitments.

Theorem 3.6.1 *(Commitment 1)*

$$\Box C1 = 0 \wedge \Box Turn = 1 \supset \Diamond\Box C2 = 1$$

In words, if C1 *is held at* 0 *and* Turn *at* 1 *then eventually* C2 *is held at* 1.

Proof: The intended interpretation is that if P1 insists on entering its critical section then eventually C2 will defer.

Either P2 halts in its non-critical section, or P2 must progress through its protocols and/or critical section. In the first case, C2 is obviously held at 1. In the second case, C2 must eventually loop indefinitely in the inner loop on Turn. Note carefully how we are using the antecedents in the formula. Since C1 and Turn *always* have a certain value, no matter what the future interleaving of instructions from the processes, P2 is constrained to follow certain branches of each test. If the antecedent only said $\Diamond C1 = 0$ or even $\Box\Diamond C1 = 0$ (which means infinitely often $C1 = 0$), we could not make this claim because the interleaving could maliciously choose to make P2 check C1 when its value was not 0. □

Theorem 3.6.2 *(Commitment 2)*

$$\Box C1 = 1 \wedge \Box Turn = 2 \supset \Diamond Turn = 1$$

In words, if C1 *is held at* 1 *and* Turn *at* 2 *then eventually* $Turn = 1$.

Proof: The intended interpretation is that if it is P2's turn to insist, eventually it will return the right to insist to P1. Do not worry about the apparent contradiction between the antecedent and the consequent concerning the value of Turn. We will use this theorem to deduce that if $\Box C1 = 1$ then indeed $\Box Turn = 2$ is not true.

This statement can be proved only under the assumption that P2 does not halt in its non-critical section. The modifications needed for that special case are simple but tedious and have been left to an exercise.

Again, the formulas in the antecedent constrain P2 to take branches in each test that lead it necessarily to its critical section. Upon exiting the critical section, P2 sets Turn to 1. Note that Turn need not be held at 1 because P1 could quickly enter its critical section and set it to 2. □

Theorem 3.6.3 *If* P1 *enters its pre-protocol, then eventually it will enter its critical section.*

Proof: We will assume that P1 does not enter its critical section and derive a contradiction.

1. $\Box Turn = 2 \supset \Diamond\Box C1 = 1$. Under the assumption that P1 does not enter its critical section, it will not exit when C2=1. Thus it must follow the other branch and then loop indefinitely in the inner loop on Turn. By the invariant relating C1 to P1's location counter, C1 is held at 1.
2. $\Box Turn = 2 \supset \Diamond Turn = 1$. Commitment 2 (3.6.2) and line 1.
3. $\neg\Box Turn = 2 \supset \Diamond Turn = 1$. If Turn is not always 2, eventually it must have some other value. From the atomic operations and the program text, this value must be 1.
4. $\Diamond Turn = 1$. Elementary propositional calculus. If p implies q and so does $\neg p$, then q is true in any case.
5. $\Diamond\Box Turn = 1$. Once $Turn = 1$, the only way that it can change is for P1 to execute its critical section and then assign 2 to Turn. But we are assuming that P1 is currently in its pre-protocol and does not enter its critical section.
6. $\Diamond\Box(at(i1) \vee at(i2))$. If Turn is held at 1 and P1 does not enter its critical section, then we have now constrained P1 to branches that cause it to loop indefinitely in its insisting loop.
7. $\Diamond\Box C1 = 0$. If P1 loops indefinitely in its insisting loop, by the invariant on C1, the value of C1 is held at 0.
8. $\Diamond\Box C2 = 1$. Lines 5 and 7 are the antecedents of commitment 1 (3.6.1), so we can conclude the consequent of that formula.

But now we have a contradiction because P1 cannot loop in its insisting loop (line 6) if the value of C2 is held at 1 (line 8). Thus we conclude that the assumption that P1 never enters its critical section is false. □

3.7 Mutual Exclusion for N Processes

Dekker's algorithm solves the mutual exclusion problem for two processes. It is possible to construct algorithms for mutual exclusion for N processes, where N is arbitrary. These solutions are not often used in practice because they are relatively complicated and in the next section we will see that simple solutions are possible with hardware assistance. Nevertheless, we will give one example of such a solution - the bakery algorithm. This algorithm is not very practical as presented, but the basic idea has been used in more advanced algorithms. In addition, it will allow us to explore an interesting variant of the common memory model of concurrency.

In the bakery algorithm, a process wishing to enter its critical section is required to take a numbered ticket whose value is greater than the values of all the

outstanding tickets. Then it waits until its ticket has the lowest value. The name of the algorithm comes from a similar situation in a bakery which has a ticket dispenser at the entrance. Customers are served in order of ascending ticket numbers.

The advantage this scheme is that there need be no variable which is both read and written by more than one process (like the variable Turn in Dekker's algorithm). Thus the bakery algorithm can be implemented on a wider range of architectures than those algorithms requiring atomic load and store to the same global variables. At the end of the section, we shall see that the bakery algorithm remains correct under an even weaker model of access to global variables.

Before looking at the full bakery algorithm, let us examine a simplified version for two processes (Figure 3.7). As promised, $N1$ is only assigned to in P1 and similarly for $N2$.

```
   N1, N2: Integer := 0;

task body P1 is
begin
  loop
a1:   Non_Critical_Section_1;
b1:   N1 := 1;
c1:   N1 := N2 + 1;
d1:   loop exit when N2 = 0 or N1 <= N2; end loop;
e1:   Critical_Section_1;
f1:   N1 := 0;
  end loop;
end P1;

task body P2 is
begin
  loop
a2:   Non_Critical_Section_2;
b2:   N2 := 1;
c2:   N2 := N1 + 1;
d2:   loop exit when N1 = 0 or N2 < N1; end loop;
e2:   Critical_Section_2;
f2:   N2 := 0;
  end loop;
end P2;
```

Figure 3.7 Simplified bakery algorithm

Theorem 3.7.1 *The algorithm satisfies the mutual exclusion property.*

Proof: We will show that the following formulas are invariant:

$$N1 = 0 \equiv at(a1) \vee at(b1) \tag{3.4}$$
$$N2 = 0 \equiv at(a2) \vee at(b2) \tag{3.5}$$

$$at(e1) \supset (at(c2) \vee N2 = 0 \vee N1 \leq N2) \tag{3.6}$$
$$at(e2) \supset (at(c1) \vee N1 = 0 \vee N2 < N1) \tag{3.7}$$

Then if $at(e1) \wedge at(e2)$, we can deduce from the above formulas that $N1 \leq N2 \wedge N2 < N1$ which is clearly impossible.

Formulas 3.4 and 3.5 are trivial to prove and 3.7 follows by symmetry from 3.6.

Formula 3.6 is trivially true initially. All transitions in P1, except for $d1 \rightarrow e1$, trivially preserve the truth of 3.6 because $at(e1)$ is false and $false \supset p$ is always true. In the case of that transition, the condition on the loop ensures that the consequent is true.

Can a transition in P2 falsify the formula?

1. $a2 \rightarrow b2$, $d2 \rightarrow d2$, $d2 \rightarrow e2$ and $e2 \rightarrow f2$ trivially preserve the truth of the invariant because they do not assign to either variable nor do they affect the truth of $at(c2)$.
2. $b2 \rightarrow c2$ makes $at(c2)$ true.
3. $c2 \rightarrow d2$ makes $N1 \leq N2$ true.
4. $f2 \rightarrow a2$ makes $N2 = 0$ true. □

The proofs of the other correctness properties (freedom from deadlock and starvation, behavior under contention) are left as exercises.

The bakery algorithm for N processes is shown in Figure 3.8. Each process chooses a number that is greater than the maximum of all outstanding ticket numbers. The array *choosing* is used as a flag to indicate to other processes that a process is in the act of choosing a ticket number. A process is allowed to enter its critical section when it has a lower ticket number than all other processes. In case of a tie in comparing ticket numbers, the lower numbered process is arbitrarily given precedence.

Theorem 3.7.2 *The bakery algorithm satisfies the mutual exclusion property.*

Proof: Assume Pi entered its critical section and then Pk entered its critical section. Before it entered its critical section, there was a last time Pi read the value of Choosing(k) and found it zero (to exit the loop $l1$), and a last time Pi read the value of Number(k) and succeeded in exiting the loop $l2$. Similarly, before Pk entered its critical section, there was a last time when it executed the three statements involved in taking a ticket.

Denoting by $read(v)$ and $write(v)$ these last times that a variable v was accessed, we have the following inequalities which are true because each process by itself is a sequential process: (3.8) describes the relations among times of instruction execution in Pi and similarly for (3.9) and Pk:[1]

$$write(N(i)) < read(C(k) = 0) < read(N(k)) \tag{3.8}$$
$$write(C(k) := 1) < read(N(i)) < write(N(k)) < write(C(k) := 0) \tag{3.9}$$

[1] We have contracted the notation and used just the first letter of each variable.

```
  Choosing: array(1..N) of Integer := (others => 0);
  Number:   array(1..N) of Integer := (others => 0);

task body Pi is
  I: constant Integer := ... ; -- Task id
begin
 -loop
    Non_Critical_Section_I;
    Choosing(I) := 1;
    Number(I)   := 1 + max(Number);
    Choosing(I) := 0;
   -for J in 1..N loop
     -if J /= I then
l1:     loop exit when Choosing(J) = 0; end loop;
l2:    -loop
          exit when
            Number(J) = 0 or
            Number(I) < Number(J) or
            (Number(I) = Number(J) and I < J);
         end loop;
       end if;
    -end loop;
    Critical_Section_I;
    Number(I) := 0;
  -end loop;
end Pi;
```

Figure 3.8 Bakery algorithm

Now in order that `Choosing(k)` have a zero value so that `Pi` can leave the loop, it must have been read either before the 1 was written, or after the 0 was written following the choice of a ticket by `Pk`. So there are two cases to consider.

$$read(C(k) = 0) < write(C(k) := 1) \tag{3.10}$$

$$write(C(k) := 0) < read(C(k) = 0) \tag{3.11}$$

Case 1: Combining the formulas (3.8 and 3.9) derived from the sequential behavior of the program with the formula (3.10) defining this case we get:

$$write(N(i)) < read(C(k) = 0) < write(C(k) := 1) < read(N(i)) \tag{3.12}$$

From (3.12) and (3.9), `Pk` read $N(i)$ and used it to compute $N(k)$ after `Pi` has chosen it current ticket number, so $N(k)$ will larger than $N(i)$. Thus `Pk` cannot enter its critical section as long as `Pi` remains in its critical section and does not assign again to $N(i)$.

Case 2: Combining the sequential formulas with (3.11) we get:

$$write(N(k)) < write(C(k) := 0) < read(C(k) = 0) < read(N(k)) \tag{3.13}$$

`Pi` read the current value $N(k)$ and then entered its critical section, proving that $N(i) < N(k)$ and thus `Pk` will not enter its critical section. □

Let us now weaken the common memory model. Rather than assume that simultaneous reads and writes to a memory location are modeled as an arbitrary interleaving of the two instructions, we allow the read to return *any* value in its range though the write is assumed to execute correctly! If there is no contention for the variable, all instructions must execute correctly.

In case 1, $write(N(k))$ in `Pk` and $read(N(k))$ in `Pi` may be executed simultaneously. This may cause `Pi` not to enter its critical section if it mistakenly read a value for $N(k)$ that is too small. However, `Pk` will correctly execute the write to $N(k)$ of a larger value than $N(i)$. On the next cycle through the loop, `Pi` will read the correct value and enter the critical section. Mutual exclusion is preserved and the simultaneous read and write only caused a short delay in one process.

Despite the elegance of the bakery algorithm it is not practical for two reasons. Firstly, the ticket numbers will be unbounded if some process is always in the critical section. Secondly, even in the absence of contention, each process must query every other process for the value of its ticket number (and in this version also for the value of *choosing*). This can be very inefficient as the number of processes increases. Thus we abandon this model for higher level models of concurrency.

3.8 Hardware-Assisted Mutual Exclusion

Under a model which can interleave individual load and store instructions, we found that it was difficult to achieve mutual exclusion. The difficulty disappears if a load and a store are allowed in a single atomic instruction.

One example is the test and set instruction. It is equivalent to the two instructions in Figure 3.9 with no interleaving allowed between them. `C` is a global

```
Li := C;
C  := 1;
```

Figure 3.9 Test and set instruction

variable with initial value 0 which is accessible to all processes. `Li` is a variable local to process `Pi`. The mutual exclusion problem for N processes can now be solved very easily as shown in Figure 3.10. We leave it as an exercise to prove that this solution satisfies the mutual exclusion property.

Another instruction is `Exchange(A,B)` which is equivalent to an atomic execution of the statements in Figure 3.11. Mutual exclusion is solved as shown in Figure 3.12 where `C` is a global variable initialized to 1. Here too, proofs of correctness are left as exercises.

3.9 Further Reading

The attempts leading up to Dekker's algorithm are taken from one of the earliest works on concurrent programming [Dij68]. A shorter algorithm is given in [Pet81]. The bakery algorithm is from [Lam74]. Raynal's book [Ray86] is a compilation of mutual exclusion algorithms.

```
task body Pi is
  Li: Integer range 0..1;
begin
  loop
    Non_Critical_Section_i;
    loop
      Test_and_Set(Li);
      exit when Li=0;
    end loop;
    Critical_Section_i;
    C := 0;
  end loop;
end Pi;
```

Figure 3.10 Mutual exclusion with Test_and_Set

```
Temp := A;
A := B;
B := Temp;
```

Figure 3.11 Exchange instruction

An algorithm by Peterson [Pet83] solves mutual exclusion for N processes using variables taking a bounded number of values (four), unlike the unbounded values of the bakery algorithm. An algorithm by Lamport [Lam87] solves the mutual exclusion problem with only five accesses to global variables per entry into the critical section in the absence of contention. Compare that to the bakery algorithm which queries every one of the other $N-1$ processes at least three times even in the absence of contention.

```
task body Pi is
  Li: Integer range 0..1 := 0;
begin
  loop
    Non_Critical_Section_i;
    loop
      Exchange(C,Li);
      exit when Li=1;
    end loop;
    Critical_Section_i;
    Exchange(C,Li);
  end loop;
end Pi;
```

Figure 3.12 Mutual exclusion with Exchange

3.10 Exercises

1. Prove the mutual exclusion property for the First Attempt. (*Hint*: The invariant is: $(at(CS1) \supset Turn = 1) \wedge (at(CS2) \supset Turn = 2)$.)
2. Prove the mutual exclusion property of Dekker's algorithm.
3. Prove the correctness properties for test and set.
4. Solve the mutual exclusion problem using `Exchange(A,B)`. Prove the correctness of the solution.
5. Prove that the values of the ticket numbers in the bakery algorithm can be unbounded.
6. Prove that the statements `Ni:=1` in the simplified bakery algorithm cannot be removed.
7. Prove that the simplified bakery algorithm is not correct under the weaker model that allows simultaneous load and store.
8. Modify the proof of the liveness of Dekker's algorithm to include the possibility that P2 is allowed to halt in its non-critical section. (*Hint*: The consequent of Commitment 2 becomes $\Diamond Turn = 1 \vee \Diamond\Box C2 = 1$. But most of the proof of Theorem 3.6.3 is devoted to using $\Diamond Turn = 1$ to prove $\Diamond\Box C2 = 1$).
9. Prove the correctness of Peterson's algorithm (Figure 3.13). (*Hint*: For liveness, prove the sequence of formulas in Figure 3.14 and show that the last two are contradictory).

```
  C1, C2: Integer range 0..1 := 0;
  Last:   Integer range 1..2 := 1;

task body P1 is
begin
  loop
a1: Non_Critical_Section_1;
b1: C1 := 1;
c1: Last := 1;
d1: loop exit when C2=0 or Last /= 1; end loop;
e1: Critical_Section_1;
f1: C1 := 0;
  end loop;
end P1;

task body P2 is
begin
  loop
a2: Non_Critical_Section_1;
b2: C2 := 1;
c2: Last := 2;
d2: loop exit when C1=0 or Last /= 2; end loop;
e2: Critical_Section_2;
f2: C2 := 0;
  end loop;
end P2;
```

Figure 3.13 Peterson's algorithm for mutual exclusion

1. $\Box(C2 = 0) \vee \Diamond(Last = 2)$
2. $at(d1) \wedge \neg\Diamond at(e1) \supset \Box\Diamond(C2 = 1)$
3. $at(d1) \wedge \neg\Diamond at(e1) \supset \Diamond(Last = 2)$
4. $at(d1) \wedge \neg\Diamond at(e1) \supset \Diamond\Box(Last = 2)$
5. $at(d1) \wedge \neg\Diamond at(e1) \supset \Box\Diamond(Last = 1)$

Figure 3.14 Proof of liveness of Peterson's algorithm

Chapter 4

Semaphores

4.1 Introduction

The algorithms in the previous chapter can be run on a *bare machine*. That is, they use only the machine language instructions that the computer provides. Though bare-machine instructions can be used to implement correct solutions to mutual exclusion and other concurrent programming problems, they are too low level to be efficient and reliable. In this chapter, we will study the *semaphore* which provides a concurrent programming primitive on a higher level than machine instructions. Semaphores are usually implemented by an underlying operating system, but for now we will investigate them by defining the required behavior and assuming that this behavior can be efficiently implemented.

A semaphore is an integer-valued variable which can take only non-negative values. Exactly two operations are defined on a semaphore S:

Wait(S) If $S > 0$ then $S := S - 1$ else suspend the execution of this process. The process is said to be suspended *on* the semaphore S.

Signal(S) If there are processes that have been suspended on this semaphore, wake one of them else $S := S + 1$.

The semaphore has the following properties:

1. `Wait(S)` and `Signal(S)` are atomic instructions.[1] In particular, no instructions can be interleaved between the test that $S > 0$ and the decrement of S or the suspension of the calling process.
2. A semaphore must be given a non-negative initial value.
3. The `Signal(S)` operation must waken one of the suspended processes. The definition does *not* specify which process will be awakened.

A semaphore which can take any non-negative value is called a *general semaphore*. A semaphore which takes only the values 0 and 1 is called a *binary semaphore* in which case `Signal(S)` is defined by: if ... else $S := 1$.

[1] The original notation is `P(S)` for `Wait(S)` and `V(S)` for `Signal(S)`, the letters P and V taken from corresponding words in Dutch.

4.2 Semaphore Invariants

A semaphore satisfies the following invariants:

$$S \geq 0 \tag{4.1}$$

$$S = S0 + \#Signals - \#Waits \tag{4.2}$$

where $S0$ is the initial value of the semaphore, $\#Signals$ is the number of signals executed on S, and $\#Waits$ is the number of completed waits executed on S. These invariants follow directly from the definition of semaphores, i.e. if you write a program and one of these formulas is not invariant, you have been given a defective implementation of semaphores.

The only non-trivial part to prove is the case of a signal which wakes a suspended process. But then $\#Signals$ and $\#Waits$ both increase by one so their difference remains invariant as does the value of S.

In the next section we give a solution to the mutual exclusion problem using semaphores and prove its correctness by appealing to the semaphore invariants.

4.3 Mutual Exclusion

Figure 4.1 is a solution to the mutual exclusion problem for two processes using semaphores. A process that wishes to enter its critical section, say P1, executes a pre-protocol that consists only of the `Wait(S)` instruction. If $S = 1$ then S can be decremented and P1 enters its critical section. When P1 exits its critical section

```
  S: Semaphore := 1;

task body P1 is
begin
  loop
    Non_Critical_Section_1;
    Wait(S);
    Critical_Section_1;
    Signal(S);
  end loop;
end P1;

task body P2 is
begin
  loop
    Non_Critical_Section_2;
    Wait(S);
    Critical_Section_2;
    Signal(S);
  end loop;
end P2;
```

Figure 4.1 Mutual exclusion with semaphores

and executes the post-protocol consisting only of the `Signal(S)` instruction, the value S will once more be 1. However, if P2 attempts to enter its critical section before P1 has left, $S = 0$ and P2 will suspend on S. When P1 finally leaves, the `Signal(S)` will wake P2.

The solution is similar to the second attempt of the previous chapter, except that the atomic implementation of the semaphore instruction prevents interleaving between the test of S and the assignment to S. It differs from the test and set instruction in that a process suspended on a semaphore no longer executes instructions checking variables in a busy-wait loop.

Theorem 4.3.1 *The mutual exclusion property is satisfied.*

Proof: Let $\#CS$ be the number of processes in their critical sections. We will prove that

$$\#CS + S = 1 \tag{4.3}$$

is invariant. Since $S \geq 0$ by invariant (4.1), simple arithmetic shows that $\#CS \leq 1$ which proves the mutual exclusion property.

To prove that (4.3) is invariant, we use the semaphore invariant (4.2).

1. $\#CS = \#Wait(S) - \#Signal(S)$. An invariant easily proven from the program text.
2. $S = 1 + \#Signal(S) - \#Wait(S)$. The semaphore invariant.
3. $S = 1 - \#CS$. From (1) and (2).
4. $\#CS + S = 1$. Immediate from (3). □

Theorem 4.3.2 *The program cannot deadlock.*

Proof: For the program to deadlock, both process must be suspended on `Wait(S)`. Then $S = 0$ because they are suspended and $\#CS = 0$ since neither is in the critical section. By the critical section invariant (4.3), $0 + 0 = 1$ which is impossible. □

Theorem 4.3.3 *There is no individual starvation.*

Proof: If P1 is suspended, the semaphore must be 0. By the semaphore invariant, P2 is in the critical section. When P2 exits the critical section, it will execute `Signal(S)` which will wake some process suspended on S. Since P1 is the only process suspended on S, it will be awakened and enter its critical section. □

Finally, it should be obvious that in the absence of contention, $S = 1$ and no single process will be delayed.

4.4 Semaphore Definitions

There are many definitions of semaphores in the literature. It is important to be able to distinguish between the various definitions because the correctness of a

program will depend on the exact definition used.

Blocked-set semaphore A signaling process awakens *one* of the suspended processes. This is the semaphore we have been using.

- `Wait(S)`: If $S > 0$ then $S := S - 1$ else suspend the execution of this process.
- `Signal(S)`: If there are processes that have been suspended on this semaphore, wake one of them else $S := S + 1$.

Blocked-queue semaphore The suspended processes are kept on a FIFO queue and awakened in the same order that they were suspended.

- `Wait(S)`: If $S > 0$ then $S := S - 1$ else suspend the execution of this process. It is appended at the tail of a FIFO queue.
- `Signal(S)`: If there are processes that have been suspended on this semaphore, wake the process at the head of the queue else $S := S + 1$.

Busy-wait semaphore The value of S is tested in a busy-wait loop. The entire if-statement is executed as an atomic operation, but there may be interleaving between cycles of the loop.

- `Wait(S)`:

```
loop
  if S > 0 then S := S-1; exit; end if;
end loop;
```

- `Signal(S)`: `S:=S+1`.

Strongly-fair semaphore If the semaphore is signaled infinitely often, eventually every waiting process will complete the semaphore instruction (that is, if l is the location of the `Wait` instruction, eventually the instruction pointer of the process will be at l': the next location following the `Wait`).

- `Wait(S)`: $at(l) \wedge \Box\Diamond S > 0 \supset \Diamond at(l')$.
- `Signal(S)`: $S := S + 1$.

Weakly-fair semaphore If the semaphore is held at a value greater than zero, eventually every waiting process will complete the semaphore instruction.

- `Wait(S)`: $at(l) \wedge \Diamond\Box S > 0 \supset \Diamond at(l')$.
- `Signal(S)`: $S := S + 1$.

The blocked-set and blocked-queue definitions define a semaphore in terms of an underlying implementation that is able to maintain data structures of suspended processes and perform actions on the data struction such as adding and deleting processes.

The busy-wait semaphore also defines a semaphore in terms of an implementation, though such an implementation is rather inefficient and can only be considered in a distributed environment where the overhead of busy-wait might not be important.

The strongly-fair and weakly-fair semaphores (or *strong* and *weak* semaphores) are defined in terms of their abstract fairness behavior and not in terms of their implementation. If a strong semaphore gets a positive value infinitely often, eventually a process that executed `Wait(S)` will complete the execution of the instruction. A weak semaphore only promises completion of `Wait(S)` if the semaphore is held indefinitely at a positive value.

A busy-wait semaphore is an implementation of a weak semaphore. It is not a strong semaphore since it is possible that S could become positive and then return to zero exactly in the intervals between the time that the process checks S. On the other hand, the blocked queue semaphore is strong since any signal will wake a blocked process.

The correctness of the solution for mutual exclusion is affected by the definition chosen for the semaphores. All semaphores satisfy the semaphore invariants, of course, so the proof of the mutual exclusion property does not change. The proof of absence of deadlock is differs only in the details. The situation is different with starvation.

First note that although the solution was given for two processes, the algorithm and the proofs immediately generalize to N processes (Figure 4.2). For this

```
S: Semaphore := 1;

task body Pi is
begin
  loop
    Non_Critical_Section_i;
    Wait(S);
    Critical_Section_i;
    Signal(S);
  end loop;
end Pi;
```

Figure 4.2 Mutual exclusion for N processes

generalization, we can prove that the starvation property of the program depends on the semaphore definition chosen.

Theorem 4.4.1 *For a busy-wait semaphore, starvation is possible.*

Proof: Consider the following execution sequence:

1. P1 executes `Wait(S)` and enters its critical section.
2. P2 finds $S = 0$ and loops.
3. P1 completes an entire cycle consisting of post-protocol, non-critical section, pre-protocol and re-enters its critical section.
4. P2 finds $S = 0$ and loops.

This can be continued indefinitely causing starvation of P2. The interleaving is conspiring to allow P2 to check the value of S exactly when P1 is in its critical section. □

Theorem 4.4.2 *For a blocked-queue semaphore, starvation is impossible.*

Proof: Suppose P1 is blocked on S. Then there are *at most* $N-1$ processes ahead of P1 on the queue for S. Thus P1 will enter its critical section after at most $N-1$ more executions of `Signal(S)`. □

Theorem 4.4.3 *For a blocked-set semaphore, starvation is possible for* $N \geq 3$.

Proof: For $N = 2$ we have shown that starvation is impossible. For $N \geq 3$, it is possible to construct an execution sequence where there will always be at least two processes blocked on S. For a blocked-set semaphore, `Signal(S)` is only required to wake one arbitrary process, so it can conspire to ignore one process causing starvation. The details of the construction are left as an exercise. □

4.5 Producer–Consumer Problem

The mutual exclusion problem is an abstraction of synchronization problems common in computer systems. Now we will look at an abstraction of communications problems called the *producer–consumer problem.* Two types of processes exist:

Producers By executing an internal procedure `Produce`, these processes create a data element which must then be sent to the consumers.

Consumers Upon receipt of a data element, these processes perform some computation in an internal procedure `Consume`.

For example, this models an application producing a listing that must be consumed by a printer process, as well as a keyboard handler producing a line of data that will be consumed by an application program. The discussion will be in terms of one producer process sending a data element to one consumer process. The generalization to multiple producers and/or multiple consumers is immediate.

Whenever a data element has to be sent from one process to another, an elementary solution is synchronous communication: when one process is ready to send and the other to receive, the data element is transferred. If more flexibility is required, a *buffer* is introduced in order to allow asynchronous communications. A buffer is a queue of data elements. The sending process *appends* a data element to the tail of the queue and the receiving process *removes* a data element from the head of a non-empty queue. The use of a buffer allows processes of similar average speeds to proceed smoothly in spite of differences in transient performance.

A familiar example is the type-ahead feature found in most interactive computer systems. The user may type several commands without waiting for the computer to complete the execution of the current command. It is possible to request six compilations and then concurrently go to lunch, which is better than entering a command every five minutes exactly. The computer is well utilized this way because it can start a second compilation immediately after finishing with the first one, rather than waiting for the user to notice that it has terminated and type in the next command.

It is important to understand that a buffer is of no use if the average speeds of the two processes are very different. To continue the example, there is no reason for a user to enter more commands than the computer can process in a single day. Conversely, if the commands are processed faster than the user can type, the buffer will remain empty.

An additional reason for using a buffer is to resolve a clash in the structure of data. A user types in characters one at a time but the computer processes complete commands. The buffer is also used to accumulate the characters typed by the user until a complete command can be submitted for processing.

4.6 Infinite Buffers

We begin the study of the producer–consumer problem with an implementation that assumes the existence of an infinite data structure, in this case an infinite array `B` that will store data of some type such as `Integer` (Figure 4.3). While

```
B: array(0..infinity) of Integer;
In_Ptr, Out_Ptr: Integer := 0;

task body Producer is
  I: Integer;
begin
  loop
    Produce(I);
    B(In_Ptr) := I;
    In_Ptr := In_Ptr + 1;
  end loop;
end Producer;

task body Consumer is
  I: Integer;
begin
  loop
    Wait until In_Ptr > Out_Ptr;
    I := B(Out_Ptr);
    Out_Ptr := Out_Ptr + 1;
    Consume(I);
  end loop;
end Consumer;
```

Figure 4.3 Producer–consumer with infinite buffer

obviously not practical, the infinite buffer leads naturally to techniques for programming finite buffers.

Two index variables `In_Ptr` and `Out_Ptr` indicate the next available position to place an element into the buffer and the oldest element which can be taken out of the buffer (see Figure 4.4).

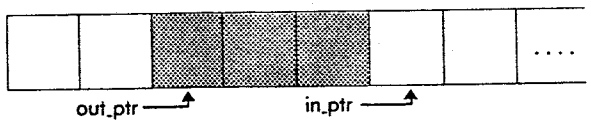

Figure 4.4 Indices in the infinite buffer

`In_Ptr` counts the number of elements placed in the buffer and `Out_Ptr` the number of elements removed. Thus $\#E = In\_Ptr - Out\_Ptr$ is the number of elements currently in the buffer and is never negative by the condition in the consumer. (Note that $In\_Ptr = Out\_Ptr$ indicates an empty buffer $\#E = 0$.) We have shown that the following equations are invariants:

$$\#E \geq 0 \tag{4.4}$$

$$\#E = 0 + In\_Ptr - Out\_Ptr \tag{4.5}$$

But these are precisely the same as equations (4.1) and (4.2) which define a semaphore! The producer–consumer problem, in particular the synchronization needed by the consumer, can be solved using a semaphore `Elements` that represents the number of elements in the buffer (Figure 4.5).

The semaphore `Elements` is explicitly counting the number of elements in the buffer. `Elements` is initialized to 0 representing the fact that the buffer is initially empty. The consumer executes `Wait(Elements)` which suspends if the number of elements in the buffer is zero. Finally we have to add a `Signal(Elements)` instruction to the producer. In the partial implementation of Figure 4.3, the consumer 'instantaneously' senses that $In\_Ptr > Out\_Ptr$. In the actual implementation, we have to physically inform the consumer whether the buffer is empty or not by signaling the semaphore.

4.7 Bounded Buffers

A practical buffer must be finite. There are two techniques used to bound the size of a buffer. The first is the *circular buffer* which is like the infinite buffer except that the index is computed modulo the length of the array (Figure 4.6). In addition to synchronization ensuring that the consumer does not try to remove an element from an empty buffer, we will have to introduce synchronization to ensure that the producer does not append to a full buffer.

The other method of bounding a buffer is to note that once data have been consumed from the low indices of the infinite buffer, that space is never used again, so we could move it to the high indices instead of asking for new array elements. This is done by allocating a *pool* of finite buffers (Figure 4.7). The producer is given a buffer to fill which is then passed to the consumer. When the consumer has removed all the elements from buffer, it is returned to the pool. Synchronization is needed to ensure that the consumer can be suspended while waiting for the next buffer from the producer and that the producer can be suspended if there


```
  B: array(0..infinity) of Integer;
  In_Ptr, Out_Ptr: Integer := 0;
  Elements: Semaphore := 0;

task body Producer is
  I: Integer;
begin
  loop
    Produce(I);
    B(In_Ptr) := I;
    In_Ptr := In_Ptr + 1;
    Signal(Elements);
  end loop;
end Producer;

task body Consumer is
  I: Integer;
begin
  loop
    Wait(Elements);
    I := B(Out_Ptr);
    Out_Ptr := Out_Ptr + 1;
    Consume(I);
  end loop;
end Consumer;
```

Figure 4.5 Semaphore solution for infinite buffers

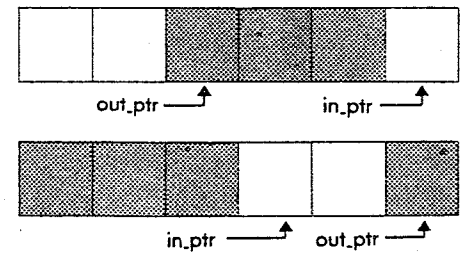

Figure 4.6 Circular buffer

are no free buffers in the pool. The circular buffer is very simple and efficient and is to be preferred in most cases. Examples of situations where a buffer pool would be more appropriate include:

- The producer or the consumer is a hardware controller that needs the data in sequential memory locations and cannot do modulo arithmetic.
- The amount of buffering needed changes with time. A communications system may need a large amount of buffer space to deal with a burst of activity on

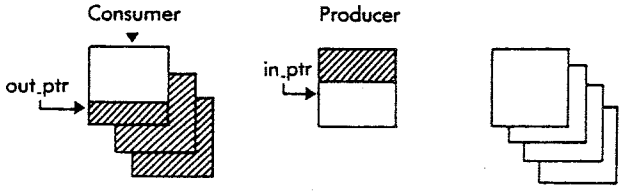

Figure 4.7 Pool of buffers

a line, but this can be reduced to almost nothing when the line is inactive. Memory utilization will be better if the buffers are shared among all the lines rather than allocating a maximal size circular buffer for each line.

- A circular buffer is a true global variable and certain architectures might find this inefficient to implement. A buffer from a pool can be copied. Also, it may be more efficient to pretend that each element created by the producer is its own buffer rather than copy the element into the circular buffer. This introduces a tight coupling between the design of the producer and the buffering scheme and should be avoided if possible.

Now we show how to program a circular buffer (Figure 4.8) and leave the buffer pool for a later chapter. Create a semaphore Spaces to count *empty* places in the buffer. When there is no more space the producer will suspend until the consumer removes an element and signals. The indices are incremented modulo N. the length of the buffer.

Theorem 4.7.1 *Let $\#E$ be the number of elements in the buffer. The following formulas are invariant at the beginning of each cycle of a loop:*

$$\#E = Elements \tag{4.6}$$

$$\#E = N - Spaces \tag{4.7}$$

Proof: Exercise. □

Theorem 4.7.2 *The program never removes an element from an empty buffer nor does it append one to a full buffer.*

Proof: Taking from an empty buffer means executing a cycle of the consumer loop when $\#E = 0$. By (4.6), $Elements = 0$ so the consumer will be suspended and not remove an element from the empty buffer. The proof for the full buffer is similar and left as an exercise. □

We also have to show that the elements are removed in the order they were appended but this proof involves techniques beyond the scope of the book.

Theorem 4.7.3 *The program cannot deadlock.*


```
   B: array(0..N-1) of Integer;
   In_Ptr, Out_Ptr: Integer := 0;
   Elements: Semaphore := 0;
   Spaces: Semaphore := N;

task body Producer is
   I: Integer;
begin
   loop
      Produce(I);
      Wait(Spaces);
      B(In_Ptr) := I;
      In_Ptr := (In_Ptr + 1) mod N;
      Signal(Elements);
   end loop;
end Producer;

task body Consumer is
   I: Integer;
begin
   loop
      Wait(Elements);
      I := B(Out_Ptr);
      Out_Ptr := (Out_Ptr + 1) mod N;
      Signal(Spaces);
      Consume(I);
   end loop;
end Consumer;
```

Figure 4.8 Semaphore solution for circular buffers

Proof: For deadlock, both processes are suspended on semaphores so $Spaces = Elements = 0$. By (4.6) and (4.7), $\#E = 0$ and $\#E = N - 0 = N$ which is impossible.[2] □

Theorem 4.7.4 *There is no starvation of either process.*

Proof: Assume the producer is suspended on `Wait(Spaces)` indefinitely while the consumer executes an infinite number of cycles of its loop. But even a single cycle of the consumer loop signals the semaphore Spaces releasing the producer. Note that like in the mutual exclusion problem, this theorem depends on the number of processes and on the definition used for the semaphores. □

[2] Impossible if $N > 0$. A buffer of zero length will cause deadlock!

4.8 Producer–Consumer with Binary Semaphores

The bounded buffer in the previous section was programmed using general semaphores which can take any non-negative value. In this section, we modify the solution to use only binary semaphores. Not only is this useful if you have a system that supplies only binary semaphores, but it also demonstrates some pitfalls that must be avoided when using these and similar primitives.

In this solution (Figure 4.9), an explicit counter `Count` is used to keep track of the number of elements in the buffer. The semaphore `S` protects this global variable from simultaneous access by the two processes and the semaphores `Not_Empty` and `Not_Full` are used to block the producer and the consumer, respectively.

Incrementing `In_Ptr` and `Out_Ptr` can be done outside the critical section, because they are used only within a single task. The load and store of the element in the buffer can also be done outside the critical section provided that the buffer is modified before incrementing or decrementing `Count`. If not, the other process could interleave statements between the update of `Count` and that of the buffer, making assumptions based on the value of `Count` that are not true of the buffer itself.

The variables `Local_Count` are used to allow a process to test the value of `Count` the last time it changed the value. This is needed because semaphores have memory. That is, for each `Signal`, there must be a corresponding `Wait`. Without the local variables (i.e. if we substitute `Count` for `Local_Count` in the test statements), the following execution sequence would be possible:

1. The producer appends an element, sets `Count` to 1, and executes `Signal (Not_Empty)`, so $Not\_Empty = 1$.
2. The consumer checks `Count` which is non-zero and does not execute `Wait (Not_Empty)`. The consumer removes an element and sets `Count` to 0. The state of the computation now is that the buffer is empty, $Count = 0$ but $Not\_Empty = 1$.
3. The consumer re-enters its loop and since $Count = 0$, it will execute `Wait (Not_Empty)`. Since $Not\_Empty = 1$, it will remove an element from an empty buffer which clearly is incorrect behavior.

The use of the local variable will cause the consumer to execute `Wait(Not_Empty)` even though this is not strictly necessary since there are actually data in the buffer.

4.9 Further Reading

This chapter is also based on [Dij68]. Stark ([Sta82]) has formalized the various definitions of semaphores and proved theorems that compare them on the mutual exclusion problem.


```
   B: array(0..N-1) of Integer;
   In_Ptr, Out_Ptr, Count: Integer := 0;
   S: Binary_Semaphore := 1;
   Not_Empty, Not_Full: Binary_Semaphore := 0;

task body Producer is
   I: Integer;
   Local_Count: Integer := 0;
begin
   loop
      Produce(I);
      if Local_Count = N then Wait(Not_Full); end if;
      B(In_Ptr) := I;
      Wait(S);
      Count := Count + 1;
      Local_Count := Count;
      Signal(S);
      if Local_Count = 1 then Signal(Not_Empty); end if;
      In_Ptr := (In_Ptr + 1) mod N;
   end loop;
end Producer;

task body Consumer is
   I: Integer;
   Local_Count: Integer := 0;
begin
   loop
      if Local_Count = 0 then Wait(Not_Empty); end if;
      I := B(Out_Ptr);
      Wait(S);
      Count := Count - 1;
      Local_Count := Count;
      Signal(S);
      if Local_Count = N-1 then Signal(Not_Full); end if;
      Out_Ptr := (Out_Ptr + 1) mod N;
      Consume(I);
   end loop;
end Consumer;
```

Figure 4.9 Circular buffer with binary semaphores

4.10 Exercises

1. Prove freedom from deadlock in the solution of the mutual exclusion problem under all definitions of semaphores.
2. Complete the proof of Theorem 4.4.3.
3. Prove Theorem 4.7.1.
4. Complete the proof of Theorem 4.7.2.

5. What would happen if several producers and consumers tried to share a buffer using the algorithm in Figure 4.8? Modify the solution to cover this case.
6. In Figure 4.9, prove that the `Count` must not be modified before `B`.
7. Prove that `Local_Count` is needed in the producer as well as the consumer of Figure 4.9.

Chapter 5

Monitors

5.1 Introduction

The semaphore was introduced to provide a synchronization primitive that does not require busy waiting. Using semaphores, we have given solutions to common concurrent programming problems. However, the semaphore is still a low-level primitive because it is unstructured. If we were to build a large system using semaphores alone, the responsibility for the correct use of the semaphores is diffused among all the implementers of the system. If one of them forgets to call `Signal(S)` after a critical section, the program can deadlock and the cause of the failure will be difficult to isolate.

Monitors provide a structured concurrent programming primitive that concentrates the responsibility for correctness into a few modules. Monitors are a generalization of the monolithic monitor (or *kernel* or *supervisor*) found in operating systems. Critical sections such as allocation of I/O devices and memory, queuing requests for I/O, and so on, are centralized in a privileged program. Ordinary programs request *services* which are performed by the central monitor. These programs are run in a hardware mode that ensures that they cannot be interfered with by ordinary programs. Because of the separation between the system and its applications programs, it is usually clear who is at fault if the system crashes (though it may be extremely difficult to diagnose the exact reason).

The monitors discussed in this chapter are decentralized versions of the monolithic monitor. Rather then have one system program handle all requests for services involving shared devices or data structures, we can define a separate monitor for each object or related group of objects. Processes request services from the various monitors. If the same monitor is called by two processes, the implementation ensures that these are processed serially to preserve mutual exclusion. If different monitors are called, their executions can be interleaved.

The syntax of monitors is based on *encapsulating* items of data and the procedures that operate upon them in a single module. The interface to a monitor will consist of a set of procedures. These procedures operate on data that are hidden within the module. The difference between a monitor and an ordinary module such as an Ada package is that a monitor not only protects internal data from

unrestricted access but also synchronizes calls to the interface procedures. The implementation ensures that the procedures are executed under mutual exclusion.

We will define a synchronization primitive that will allow a process to suspend itself if necessary. For example, in the producer–consumer problem:

- The only operations permitted on a buffer are append and remove an item.
- Append and remove exclude each other.
- A producer will suspend on a full buffer and a consumer on an empty buffer.

5.2 Producer–Consumer Problem

We will define the monitor construct in parallel with the solution of the producer–consumer problem (Figure 5.1).[1] Note that the monitor is not a process (Ada task), but a static module of data and procedure declarations. The actual producer and consumer processes have to be programmed separately (Figure 5.2).

```
monitor Producer_Consumer_Monitor is
  B: array(0..N-1) of Integer;
  In_Ptr, Out_Ptr: Integer := 0;
  Count: Integer := 0;
  Not_Full, Not_Empty: Condition;

procedure Append(I: in Integer) is
begin
    if Count = N then Wait(Not_Full); end if;
    B(In_Ptr) := I;
    In_Ptr := (In_Ptr + 1) mod N;
    Signal(Not_Empty);
end Append;

procedure Take(I: out Integer) is
begin
    if Count = 0 then Wait(Not_Empty); end if;
    I := B(Out_Ptr);
    Out_Ptr := (Out_Ptr + 1) mod N;
    Signal(Not_Full);
end Take;

end Producer_Consumer_Monitor;
```

Figure 5.1 Monitor for producer-consumer

Despite the syntactic similarity to an ordinary module (Ada package), the semantics of a monitor are different because only one process is allowed to execute

[1] Unlike most of the examples in this book, this one is not executable in Ada without modification. See Appendix B for details.


```
task body Producer is
  I: Integer;
begin
  loop
    Produce(I);
    Append(I);
  end loop;
end Producer;

task body Consumer is
  I: Integer;
begin
  loop
    Take(I);
    Consume(I);
  end loop;
end Consumer;
```

Figure 5.2 Producer and consumer processes

a monitor procedure at any time. In this case, the producer can be executing Append or the consumer Take, but not both. This ensures the mutual exclusion on the global variables, in particular, on the variable Count which is updated by both procedures.

The solution is more structured than the semaphore solution both because the data and procedures are encapsulated in a single module and because the mutual exclusion is provided automatically by the implementation. The producer and consumer processes see only abstract Append and Take operations and do not have to be concerned with correctly programming semaphores.

The solution that used binary semaphores, used three of them: S for mutual exclusion, and Not_Empty and Not_Full for synchronization. The mutual exclusion requirement is now satisfied by the definition of monitors. For synchronization, we define a structure called *condition variables*. A condition variable C has three operations defined upon it:[2]

Wait(C) The process that called the monitor procedure containing this statement is suspended on a FIFO queue associated with C. The mutual exclusion on the monitor is released.

Signal(C) If the queue for C is non-empty then wake the process at the head of the queue.

Non_Empty(C) A boolean function that returns true if the queue for C is non-empty.

The Wait operation allows a process to suspend itself. Conventionally, the name of the condition variable is chosen so that Wait(C) can be read: 'I am waiting

[2] We are using the same names *Wait* and *Signal* that were used for the semaphore operations, but there is no relation between the two primitives.

for C to occur', and Signal(C): 'I am signaling that C has occurred'. However, a condition variable is just a signaling device, unlike a general semaphore which has a counter associated with it, so it is the responsibility of the programmer to ensure that the condition defined by C has actually occurred.

Wait(C) releases the mutual exclusion on the monitor, so that other processes can enter (and presumably help establish the condition required by C). Signal(C) is *memoryless*, that is, if no process is suspended on the queue for C, the operation is ignored.

In the producer–consumer example, we use two condition variables:

Not_Empty Used by the consumer to suspend itself until the buffer is not empty.

Not_Full Used by the producer to suspend itself until the buffer is not full.

The definition of Signal(C) only requires it to wake the first process suspended on C. It does not require this process to actually be scheduled for execution. In fact, the problem is not simple for the following reason. Remember that the process executing Signal(C) is inside a monitor procedure. Since we cannot have two processes within a monitor, this seems to require that the awakened process remain suspended until the signaling process has exited the monitor. Then, perhaps, we could require that the awakened process have priority over other processes. But what would happen if the signaling process issued more than one signal?

The difficulty is more than a matter of finding and implementing a reasonable definition for the signal operation. Remember that a signal is issued when a process has determined that the condition required by the suspended process exists. If we allow execution of any instructions between the signal and the time that the awakened process is allowed to continue, we will have to verify that the condition continues to hold.

The problem can be demonstrated in the producer–consumer problem. Suppose that there were two consumer processes C1 and C2 and one producer process P1. C1 is suspended on Not_Empty, C2 has just called Take and P1 is about to execute Signal(Not_Empty). P1 awakens C1 and the buffer is not empty, but if C2 is allowed to enter the monitor, it could make the buffer empty again before C1 has a chance to act. Once C2 leaves the buffer, C1 is allowed to continue and will incorrectly take a value from an empty buffer.

The solution is to require that an awakened process be scheduled for immediate execution after the signal instruction. This *immediate resumption requirement* simplifies the programming and verification of software that uses monitors. This takes care of the problem demonstrated in the previous paragraph by giving priority to awakened processes over those that have yet to enter the monitor. There remains the problem of defining the relative priority of the signaling process and the awakened process. The simplest solution is to require that a signal be the *last* instruction executed in the procedure in which it occurs. Then the execution of the signal can be considered to reschedule the awakened process.

There are several ways of avoiding this restriction on signals, but we will not discuss them in the text.

5.3 Emulation of Semaphores by Monitors

In this section and the next one, we will show how to emulate a semaphore using monitors and conversely. This will not only show that the two primitives are of similar power and expressibility, but will also show that the monitor is a higher-level abstraction than a semaphore because the emulation in one direction will be so much easier. These emulations can also be used to port a program from a system supplying one primitive to a system supplying the other one.

```
monitor Semaphore_Emulation is
  S: Integer := S0;
  Not_Zero: Condition;

  procedure Semaphore_Wait is
  begin
    if S=0 then Wait(Not_Zero); end if;
    S := S - 1;
  end Semaphore_Wait;

  procedure Semaphore_Signal is
  begin
    S := S + 1;
    Signal(Not_Zero);
  end Semaphore_Signal;
end monitor;
```

Figure 5.3 Emulation of semaphores by monitors

The monitor in Figure 5.3 emulates a semaphore. The variable S holds the value of the semaphore and is initialized to some non-negative value S0. (We could also have defined another procedure to initialize S.) The condition variable Not_Zero maintains the queue of processes waiting for the semaphore to be non-zero.

Theorem 5.3.1 *The semaphore invariants hold:*

$$S \geq 0 \tag{5.1}$$

$$S = S0 + \#waits - \#signals \tag{5.2}$$

Proof: As usual, the proof is by induction on the execution sequence. Since each monitor procedure is executed under mutual exclusion with no possibility of interleaving, we can relax the proof rules. It is sufficient to prove that the formulas are invariant in any interleaving where every execution of a monitor procedure is a single atomic instruction. The fact that the variables may temporarily have values that falsify the invariant may be ignored since no other process can see these values. Remember that a Wait(C) instruction is considered to cause a process to leave the monitor, so the invariants must be checked there too.

For (5.1), $S0$ is assumed non-negative, so (5.1) is initially true. The only transition that could falsify the formula is an execution of `Semaphore_Wait` with $S = 0$. In that case, `Wait(Non_Zero)` will suspend the process. It will be awakened only by an execution of `Signal(Non_Zero)`. But the previous statement in `Semaphore_Signal` increased the value of `S` to 1. By the immediate resumption requirement, $S = 1$ when it is decremented.

The proof of 5.2 is left as an exercise. □

This proves the safety properties required of an implementation of a semaphore. The emulation implements a block-queue semaphore because condition variables are defined to have FIFO queues and because the immediate resumption requirement ensures that there is no possibility of interleaving a third process between a signaling process and the process awakened from a suspend wait.

5.4 Emulation of Monitors by Semaphores

To emulate monitors using semaphores, we need a semaphore `S` to ensure mutual exclusion on the monitor procedures and one semaphore `C_Semaphore` for each condition variable. The semaphores will be assumed to be blocked-queue semaphores in order to implement the FIFO requirement of monitors. Otherwise, we would have to program explicitly a queue of blocked processes.

In addition, we need a counter for each condition variable because the meaning of `Signal(C)` depends on whether the queue is empty or not. But there is no way to discover this for a semaphore, so we will have to explicitly program the counter.

Each procedure in the monitor will have `Wait(S)` as its first instruction and `Signal(S)` as its last.[3]

Each occurrence of `Wait(C)` is translated to:

```
C_Count := C_Count + 1;
Signal(S);
Wait(C_Semaphore);
Wait(S);
C_Count := C_Count - 1;
```

The definition of `Wait(C)` requires that the mutual exclusion on the monitor be released by `Signal(S)`. Obviously, to avoid deadlock, this must be done before waiting for the condition.

Each occurrence of `Signal(C)` is translated to:

```
if C_Count > 0 then
  Signal(C_Semaphore);
end if;
```

However, this does not correctly emulate the immediate resumption requirement. Remember that we have assumed that a `Signal(C)` will be the last instruction in a procedure, so a `Signal(S)` will follow immediately:

[3] Be careful not to confuse waits and signals of the two primitives.


```
if C_Count > 0 then
  Signal(C_Semaphore);
end if;
Signal(S);
```

Signaling the condition semaphore will awaken the suspended process. But it is still possible to interleave instructions from a third process between `Wait(C_Semaphore)` and `Wait(S)`. This third process could execute `Wait(S)` before the awakened process and violate the immediate resumption requirement. A signal on a blocked-queue semaphore is only required to awaken the process at the head of the queue in preference to other suspend processes; it is not required to give it priority scheduling.

The solution is not to release the mutual exclusion on the monitor when signaling, but to let it remain in force and have the awakened process take upon itself the requirement for releasing mutual exclusion. The pair of instructions `Signal(C)` followed by the end of the monitor procedure is now translated:

```
if C_Counter > 0 then
  Signal(C_Semaphore);
else
  Signal(S);
end if;
```

and `Wait(C)` is simplified to:

```
C_Count := C_Count + 1;
Signal(S);
Wait(C_Semaphore);
C_Count := C_Count - 1;
```

5.5 The Problem of the Readers and the Writers

The problem of the readers and the writers is the next abstract problem in concurrent programming that we will solve. It is similar to the mutual exclusion problem in that several processes are competing for access to a critical section. In this problem, however, we divide the processes into two classes:

Readers Processes which are not required to exclude one another.

Writers Processes which are required to exclude every other process, readers and writers alike.

The problem is an abstraction of access to databases, where there is no danger in having several processes read concurrently, but writing or changing the data must be done under mutual exclusion to ensure consistency. The concept database must be understood in the widest possible sense – even a simple table describing the status of a disk drive or the arrangement of windows on a terminal screen can be considered a database that may need to be protected by some solution to the problem of the readers and the writers.

A process that wishes to read (write) calls monitor procedure `Start_Read` (`Start_Write`). Upon returning from the procedure, the process reads (writes) and then calls another monitor procedure `End_Read` (`End_Write`) to indicate to the

monitor that it has terminated (Figure 5.4). Even before looking at the monitor itself, it should be clear that the start-procedures may cause a process to suspend but that the end-procedures will only signal.

```
task body Reader is
begin
  loop
    Start_Read;
    Read_the_Data;
    End_Read;
  end loop;
end Reader;

task body Writer is
begin
  loop
    Start_Write;
    Write_the_Data;
    End_Write;
  end loop;
end Writer;
```

Figure 5.4 Readers and writers

The monitor (Figure 5.5) has two status variables:

Readers A counter of the number of readers which have successfully passed `Start_Read` and are currently reading.

Writing A flag which is true when a process is writing.

There are two condition variables:

OK_to_Read to suspend readers.

OK_to_Write to suspend writers.

The general form of the monitor code is not difficult to follow. The variables `Readers` and `Writing` are incremented[4] in the start-procedures and decremented in the end-procedures. At the beginning of the start-procedures, a boolean expression is checked to see if the process should be suspended and at the end of the end-procedures, an expression is checked to see if some condition should be signaled. These expressions determine the behavior of the monitor.

A reader is suspended if some process is currently writing ($Writing$) or is some process is waiting to write ($Non\_Empty(OK\_to\_Write)$). The first condition is obviously required by the statement of the problem. The second condition is a decision to give the first suspended writer priority over waiting readers. On the other hand, the writer is suspended only if there are processes currently reading

[4] It is convenient to use the term 'increment' for the boolean variable too, where `True` is greater than `False`.

($Readers \neq 0$) or writing ($Writing$). `Start_Write` does not check the condition queues.

`End_Read` executes `Signal(OK_to_Write)` if there are no more readers. If there are suspended writers, one will be awakened and allowed to complete `Start_Write`. Otherwise, the operation does nothing and we return to the initial state. `End_Write` gives priority to the first suspended reader, if any, otherwise it wakes suspended writers, if any.

Finally, what is the function of `Signal(OK_to_Read)` in `Start_Read`? This statement performs a *cascaded wakeup* of the suspended readers. Upon termination of a writer, we gave priority to waking a suspended reader over a suspended writer. However, if one process is allowed to read, we might as well awaken them all. When the first reader completes `Start_Read`, it will signal the next reader and so on, until this cascade of signals wakes all the currently suspended readers.

What about readers that attempt to start reading during the cascaded wakeup? Will they have priority over suspended writers? By the immediate resumption requirement, the cascaded wakeup will run to completion before any new reader is allowed to commence execution of a monitor procedure. When the last `Signal(OK_to_Read)` is executed (and does nothing because the queue is empty), the monitor will be released and a new reader may enter. However, it is subject to the usual check that will cause it to suspend if there are waiting writers.

To summarize:

- If there are suspended writers, a new reader is required to wait until the termination of (at least) the first write.
- If there are suspended readers, they will (all) be released before the next write.

5.6 Correctness Proofs

This section contains proofs of the correctness of the solution to the problem of the readers and the writers given in the previous section.

Let R be the number of processes currently reading and W the number of processes currently writing. Then the following formulas are invariant:[5]

$$R = Readers \tag{5.3}$$

$$W > 0 \equiv Writing \tag{5.4}$$

$$Non\_Empty(OK\_to\_Read) \supset (Writing \vee Non\_Empty(OK\_to\_Write)) \tag{5.5}$$

$$Non\_Empty(OK\_to\_Write) \supset (Readers \neq 0 \vee Writing) \tag{5.6}$$

The proofs are left as exercises. They are similar to the proofs of the correctness of the semaphore invariants for the emulation of semaphores by monitors. Remember that monitor invariants are only required to hold outside the monitor

[5] Note that `Writing` is a boolean variable so its value $Writing$ is a boolean expression which is either $True$ or $False$. Similarly for the boolean function `Non_Empty`.

```
monitor Reader_Writer_Monitor is
  Readers: Integer := 0;
  Writing: Boolean := False;
  OK_to_Read, OK_to_Write: Condition;

procedure Start_Read is
begin
  if Writing or Non_Empty(OK_to_Write) then
    Wait(OK_to_Read);
  end if;
  Readers := Readers + 1;
  Signal(OK_to_Read);
end Start_Read;

procedure End_Read is
begin
  Readers := Readers - 1;
  if Readers = 0 then Signal(OK_to_Write); end if;
end End_Read;

procedure Start_Write is
begin
  if Readers /= 0 or Writing then
    Wait(OK_to_Write);
  end if;
  Writing := True;
end Start_Write;

procedure End_Write is
begin
  Writing := False;
  if Non_Empty(OK_to_Read) then
    Signal(OK_to_Read);
  else
    Signal(OK_to_Write);
  end if;
end End_Write;
end Reader_Writer_Monitor;
```

Figure 5.5 Monitor for readers and writers

procedures themselves and that the immediate resumption requirement allows one to infer that what was true before executing a signal is true upon release of a wait.

Theorem 5.6.1 *(Safety property of the solution) The following formula is invariant:*

$$(R > 0 \supset W = 0) \wedge (W > 0 \supset (W = 1 \wedge R = 0)) \tag{5.7}$$

In words: if some process is reading, there are no writers and if there are writers, there is only one writer and, furthermore, there are no readers.

Proof: Formula 5.7 is initially true since $R = W = 0$. Assume that 5.7 is true and show that each transition preserves the truth of the formula. The formula can be falsified by falsifying either of its conjuncts, so it is sufficient to show that no transition can falsify either one or the other.

Case 1: First conjunct:

$$R > 0 \supset W = 0 \tag{5.8}$$

Suppose that both antecedent ($R > 0$) and consequent ($W = 0$) are true. Formula 5.8 could be falsified if a process began to write, which can only happen if it successfully completes `Start_Write`.

Subcase 1.1: The process enters `Start_Write` and successfully completes it. But we are assuming that $R > 0$ and by (5.3), *Readers* > 0. Thus the process cannot complete `Start_Write`, instead it will be suspended on `Wait(OK_to_Write)`.

Subcase 1.2: The process was suspended on `OK_to_Write` and was awakened by a signal. As before, from $R > 0$ we can deduce that *Readers* > 0, so the signal in `End_Read` will not be executed. Since $W = 0$, there are no writers, so no writer will executed the signal in `End_Write`.

Case 2: As in case 1, but assume that both $R > 0$ and $W = 0$ are false and that a process begins to read causing only the antecedent to become true.

Subcase 2.1: $\neg W = 0$ implies *Writing* by (5.4) so a process trying to execute `Start_Read` will be suspended.

Subcase 2.2: From $\neg R > 0$, we deduce that there are no readers, so no reader will execute the signal in `Start_Read`. Since (5.7) is assumed true, $\neg W = 0$ implies $W = 1$. Then `Signal(OK_to_Read)` in `End_Write` will be executed causing $R > 0$ to become true, but $W = 0$ will also become true. Thus it is impossible to falsify the antecedent alone.

Case 3: Second conjunct:

$$W > 0 \supset (W = 1 \wedge R = 0) \tag{5.9}$$

Assume that the antecedent and consequent are both true and that the consequent is falsified. This can happen if a second process begins to write or if some process begins to read. The proofs that this is impossible are similar to the previous cases and are left to the reader.

Case 4: As in case 3, but assume that the antecedent and consequent are false and that the antecedent alone becomes true. As in Subcase 2.2, to go from $\neg W > 0$ to $W > 0$ means that W was 0 and now is 1. Also, $R = 0$ is true by the code in `Start_Write` for the subcase of a process entering and successfully exiting the procedure. For the subcase of a writer being awakened, we leave it as an exercise using (5.6). □

Theorem 5.6.2 *No process is starved.*

Proof: We will prove that no process wishing to read is starved and leave the proof for writers as an exercise.

If a process P tries to execute `Start_Read`, there are three possible outcomes:

1. P successfully completes the execution of the procedure. In this case, there is no starvation.
2. P never enters the procedure. Every other process either completes a monitor procedure or is suspended, waiting for a condition which also frees the mutual exclusion on the monitor. Since there is are a finite number of processes, eventually P will be allowed to enter.[6] This case reduces to the other two.
3. P is suspended by `Wait(OK_to_Read)`. This is the difficult case.

So suppose that P is enqueued on `OK_to_Read`. There are only a finite number of processes enqueued before P. We fill show that eventually `Signal(OK_to_Read)` must be executed. By (ordinary numerical) induction on the number of processes before P we can conclude that there is no starvation.

By invariant 5.5, *Writing* or *Non_Empty*(*OK_to_Write*).

Case 1: Writing. By (5.4), $W > 0$. Eventually one of these writers (actually there is only one) must finish writing and execute `End_Write`. Since *Non_Empty (OK_to_Read)*, the writer must execute `Signal(OK_to_Read)`.

Case 2: Non_Empty(*OK_to_Write*). By (5.6), *Readers* $\neq 0 \vee$ *Writing*. If *Writing*, then eventually a writer executes `Signal(OK_to_Read)` as in case 1. If *Readers* $\neq 0$, eventually they each execute `End_Read`. Since *Non_Empty (OK_to_Write)*, no new readers can complete `Start_Read`, so eventually the last reader executes `Signal(OK_to_Write)` in `End_Read` and this reduces to the previous cases. □

5.7 Further Reading

Monitors were introduced by Hoare [Hoa74]. The verification techniques are from [How76]. Monitors are extensively used in concurrent programming. [BH77] and [BEW88] describe Pascal dialects that include monitor-based concurrent programming primitives. The books also include complete examples of systems written in these languages.

5.8 Exercises

1. Prove that the emulation of semaphores by monitors gives a blocked-queue semaphore.
2. Prove that the emulation of monitors by semaphores is no longer correct if blocked-set semaphores are used.
3. Program and prove the correctness of modifications to the solution to the problem of the readers and the writers which implement the following rules:
   - If there are reading processes, a new reader may commence reading even if there are waiting writers.

[6] There is an implicit assumption that the data structure containing the processes trying to enter the monitor is also a FIFO queue.

   - If there are waiting writers, they receive priority over all waiting readers.
   - If there are waiting readers, no more than two writers will write before a process is allowed to read.

Chapter 6

The Problem of the Dining Philosophers

6.1 Introduction

The problem of the dining philosophers is a classic in the field of concurrent programming. While of little intrinsic interest, it offers an entertaining vehicle for comparing various formalisms for writing and proving concurrent programs. It is sufficiently simple to be tractable yet subtle enough to be challenging.

The problem is set in a secluded community of five philosophers. The philosophers engage in only two activities - *thinking* and *eating* (Figure 6.1).

```
task body Philosopher is
begin
  loop
    Think;
    Pre_Protocol;
    Eat;
    Post_Protocol;
  end loop;
end Philosopher;
```

Figure 6.1 Form of dining philosopher solution

Meals are taken communally at a table set with five plates and five forks (Figure 6.2). In the center of the table is a bowl of spaghetti that is endlessly replenished. Unfortunately, the spaghetti is hopelessly tangled and a philosopher needs two forks in order to eat.[1] Each philosopher may pick up the forks on his left and right, but only one at a time. The problem is to design pre- and post-protocols to ensure that a philosophers only eats if he has two forks. The solution should also satisfy the usual correctness properties that we described in the chapter on mutual exclusion. The set of correctness properties are:

[1] It is part of the folklore of computer science to point out that the story would be more believable if the bowl contained rice and the utensils were chopsticks.

1. A philosopher eats only if he has two forks.
2. No two philosophers may hold the same fork simultaneously.
3. No deadlock.
4. No individual starvation (pun intended).
5. Efficient behavior under absence of contention.

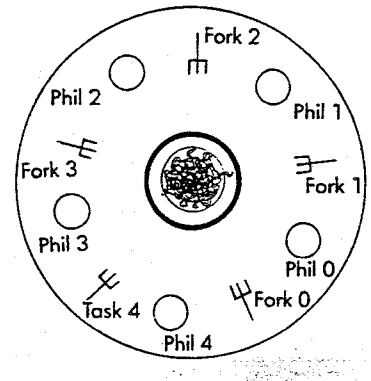

Figure 6.2 The dining table

6.2 Solutions using Semaphores

A first attempted solution is shown in Figure 6.3. We assume that each philosopher is initialized with its index I. Each fork is modeled as a semaphore. A philosopher must complete a `Wait` on both his left- and right-hand forks before eating.

```
  Fork: array(0..4) of Semaphore := (others => 1);

task body Philosopher is
begin
  loop
    Think;
    Wait(Fork(I));
    Wait(Fork((I+1) mod 5));
    Eat;
    Signal(Fork(I));
    Signal(Fork((I+1) mod 5));
  end loop;
end Philosopher;
```

Figure 6.3 First attempt

Theorem 6.2.1 *No fork is ever held by two philosophers.*

Proof: The statement `Eat` is a critical section for each fork. If $\#Pi$ is the number of philosophers holding fork i then we have:

$$Fork(I) + \#Pi = 1$$

Together with the invariant that a semaphore is non-negative, we can conclude that $\#Pi \leq 1$. □

Unfortunately, this solution deadlocks under an interleaving that has all philosophers pick up their left forks (i.e. execute `Wait(Fork(I))`) one after another before any of them tries to pick up a right fork. Now they are all waiting on their right forks, but no process will ever signal.

One way of ensuring liveness in a solution to the dining philosophers problem is to limit the number of philosophers entering the dining room to four (Figure 6.4).

```
  Room: Semaphore := 4;
  Fork: array(0..4) of Semaphore := (others => 1);

task body Philosopher is
begin
  loop
    Think;
    Wait(Room);
    Wait(Fork(I));
    Wait(Fork((I+1) mod 5));
    Eat;
    Signal(Fork(I));
    Signal(Fork((I+1) mod 5));
    Signal(Room);
  end loop;
end Philosopher;
```

Figure 6.4 Limit the number of philosophers

The addition of the `Room` semaphore obviously does not affect the correctness of the safety properties we showed in the last example.

Theorem 6.2.2 *Individual starvation cannot occur.*

Proof: We must assume that the `Room` semaphore is a blocked-queue semaphore so that any philosopher waiting to enter the room will eventually do so. The `Fork` semaphores need only be blocked-set semaphores since only two philosophers use each one.

Suppose philosopher i is starved. Then he is blocked forever on a semaphore. There are three cases depending on whether he is blocked on `Room`, `Fork(i)` or `Fork(i+1)`.[2]

Case 1: By the assumption that the semaphore is FIFO, i is blocked on `Room` only if its value is 0 indefinitely. By the semaphore invariant, this can happen only if the other four philosophers are blocked on forks, because if one of them should raise two forks and eat it will eventually finish eating, put down the forks and signal `Room`. Thus this case follows from the other two. Note that eating is considered to be a critical section which will eventually terminate.

Case 2: i is blocked on his left fork. Then philosopher $i-1$ holds `Fork(i)` as his right fork. That is, philosopher $i-1$ has successfully executed his last `Wait` statement and is either eating or signaling, both of eventually terminate releasing i.

Case 3: i is blocked on his right fork. This means that philosopher $i+1$ has successfully taken his left fork (`Fork(i+1)`) and never released it. Since neither eating nor signaling can block, $i+1$ must be suspended forever on his right fork. By induction, it follows that if i is blocked on his right fork, then so must all philosophers: $i+j, 0 \leq j \leq 4$. However, by the semaphore invariant on `Room`, for some $1 \leq j \leq 4$, philosopher $i+j$ is not in the room and thus obviously not blocked on a fork semaphore. □

Theorem 6.2.3 *Deadlock cannot occur.*

Proof: Immediate. □

Another solution that is free from starvation is the asymmetric solution (Figure 6.5) which has the first four philosophers execute the original solution, but the fifth philosopher waits first for the right fork and then for the left fork. Again it is obvious that the correctness properties on eating are satisfied. There cannot be deadlock or starvation. The proofs are similar to the previous one and are left as an exercise.

```
task body Philosopher is
begin
    Think;
    Wait(Fork(0));
    Wait(Fork(4));
    Eat;
    Signal(Fork(4));
    Signal(Fork(0));
  end loop;
end Philosopher;
```

Figure 6.5 Asymmetric solution to dining philosophers

[2] The computation of the index modulo 5 will not be explicitly noted in the proof.

6.3 Monitor Solutions to the Dining Philosophers

The difficulty with finding a semaphore solution to the problem of the dining philosophers is caused by the fact that executing a Wait on a semaphore is irrevocable. There is no way to test the value of two fork semaphores simultaneously. Using the monitor primitive, we can find an elegant solution by having a philosopher wait until both forks are free. Figure 6.6 is a monitor controlling the forks. Philosophers call the monitor as shown in Figure 6.7 to request forks. The process will leave the monitor only if it has two forks.

```
monitor Fork_Monitor
  Fork: array(0..4) of Integer range 0..2:=(others=>2);
  OK_to_Eat: array(0..4) of Condition;

procedure Take_Fork(I: Integer) is
begin
  if Fork(I) /= 2 then Wait(OK_to_Eat(I)); end if;
  Fork((I+1) mod 5) := Fork((I+1) mod 5)-1;
  Fork((I-1) mod 5) := Fork((I-1) mod 5)-1;
end Take_Fork;

procedure Release_Fork(I: Integer) is
begin
  Fork((I+1) mod 5) := Fork((I+1) mod 5)+1;
  Fork((I-1) mod 5) := Fork((I-1) mod 5)+1;
  if Fork((I+1) mod 5)=2 then
    Signal(OK_to_Eat((I+1) mod 5));
  end if;
  if Fork((I-1) mod 5)=2 then
    Signal(OK_to_Eat((I-1) mod 5));
  end if;
end Release_Fork;
end Fork_Monitor;
```

Figure 6.6 Monitor for dining philosophers

The monitor maintains an array Fork which counts the number of free forks available to each philosopher. The Take_Fork procedure waits on its own condition variable until two forks are available. Before leaving the monitor with these two forks, it decrements the number of free forks available to its neighbors. After eating, a philosopher calls Release_Fork. In addition to updating the available fork array, the procedure checks if freeing the forks makes it possible to signal a neighbor.

Theorem 6.3.1 *A philosopher eats only if he has two forks.*

Proof: It is an invariant that $Eating(I) \supset Fork(I) = 2$. Remember that a monitor invariant need only be true outside the monitor procedures. Then use


```
task body Philosopher is
begin
  loop
    Think;
    Take_Fork(I);
    Eat;
    Release_Fork(I);
  end loop;
end Philosopher;
```

Figure 6.7 Philosopher processes calling the monitor

the program code and the immediate resumption requirement. The details are left as an exercise. □

Theorem 6.3.2 *The solution does not deadlock.*

Proof: Let *eating* be the number of philosophers who have successfully completed Take_Fork to commence eating. Then the following two formulas can be proved invariant by the techniques used in the chapter on monitors:

$$Non\_Empty(OK\_to\_Eat(i)) \supset Fork(i) < 2 \tag{6.1}$$

$$\sum_{i=0}^{4} Fork(i) = 10 - 2 * Eating \tag{6.2}$$

Deadlock implies $Eating = 0$ and all philosophers are enqueued on OK_to_Eat. If no philosophers are eating, from (6.2) we conclude $\sum Fork(i) = 10$. If they are all enqueued waiting to eat, from (6.1) we conclude $\sum Fork(i) \leq 5$ which gives a contradiction in the value of $Fork$. □

However, this solution can cause starvation if two philosophers conspire to starve their mutual neighbor (Figure 6.8). Philosophers 1 and 3 both need a fork belonging to philosopher 2. If 1 finishes eating and puts fork 2 down, 2 cannot

Action	Fork(0)	Fork(1)	Fork(2)	Fork(3)	Fork(4)
(Initially)	2	2	2	2	2
Take_Fork(1)	1	2	1	2	2
Take_Fork(3)	1	2	0	2	1
Take_Fork(2) and Wait(OK_to_Eat(2))	1	2	0	2	1
Release_Fork(1)	2	2	1	2	1
Take_Fork(1)	1	2	0	2	1
Release_Fork(3)	1	2	1	2	2
Take_Fork(3)	1	2	0	2	1

Figure 6.8 Starvation of philosopher 2

start until 3 also finishes. If 1 tries to eat again before 3 has finished, he will find fork 2 still available and commence eating. Now philosopher 3 does the same thing and this interleaving can be continued indefinitely.

6.4 Further Reading

The dining philosophers problem has become a classic example ever since it was introduced by Dijkstra [Dij71].

Perfectly symmetrical solutions to problems in concurrent programming are impossible because if every process executes exactly the same program, they can never 'break ties'. All our algorithms contain asymmetries in the form of process identifiers or a kernel maintaining a queue. An interesting paper by Chandy and Misra [CM84] describes a starvation-free algorithm where precedence among the processes is indicated by a directed acyclic graph (see also [CM88]). The sink node with no outgoing edges has the highest priority. When that process has eaten it changes the graph to increase the precedence of other processes. After the modifications the graph is still acyclic. Though explicit process identifiers are no longer needed, the acyclic graph implicitly decides relative precedence.

6.5 Exercises

1. Prove the correctness of the asymmetric solution.
2. Investigate the dining philosophers problem for 2, 3 and 4 philosophers.
3. Program a solution that suffers from livelock. (*Hint:* A philosopher picks up his left fork and then checks if his right fork is free. If not he puts down his left fork and tries again.)

PART II

Distributed Programming

Chapter 7

Distributed Programming Models

7.1 Introduction

The concurrent programming abstractions of Part I are based on mutual exclusion on the access to an individual memory location or a system routine. Part II raises the level of abstraction to processes which *communicate* by sending *messages* to each other. Sending a message is a higher level action that can be easily implemented on physically distributed processors, hence the name given to this field: *distributed programming* or *distributed processing*.

As before, we will be dealing with an abstraction and not with the underlying implementation. In a physically distributed system, messages are exchanged along *communications channels* using *protocols* that define the format of a message as well as procedures for error detection and correction. The construction of such computer networks is beyond the scope of this book. Instead, our primitive notion will be that of sending and receiving a message. Unless otherwise indicated, we assume that a message arrives correctly – any error correction is done by the underlying protocol. Again, absolute time will not be used and messages may be delayed arbitrarily.

The high-level abstractions of distributed programming models are also used on single-processor computers, so the phrase 'send a message' need not imply the existence of a computer network. It can be implemented by calling a routine in an underlying operating system.

When designing primitives for distributed programming, there are a wide variety of decisions that must be made. The rest of this introductory chapter sets out the various options. The next three chapters describe languages – Ada, occam and Linda – whose designs are radically different on these points. Then we present some classical algorithms that are based on distributed programming models.

7.2 Synchronous or Asynchronous Communication

The common-memory models are based on regulating contention for a resource like a memory location or a monitor procedure. In the absence of contention, a single process need not be delayed. Communication, however, requires two

processes - one to send a message and one to receive it. When a process sends a message, we must decide if the receiver must co-operate, that is if the receiver must be ready to receive the message when it is sent, or if the process can send one or more messages regardless of the state of the receiver.

In *synchronous communication* the exchange of a message is an atomic action requiring the participation of both the sending process - the *sender* - and the receiving process - the *receiver*. If the sender is ready to send but the receiver is not ready to receive, the sender is blocked and similarly, if the receiver is the first process ready to communicate, it blocks. The act of communication synchronizes the execution sequences of the two processes. The term *rendezvous* is often used to evoke an image of two processes that have to get to a designated point. The first process to arrive must wait for the second.

Alternatively, the sender is allowed to send a message and continue without blocking. The communication is called *asynchronous* because there is no time connection between the execution sequences of the two processes. The receiver could be executing any instructions when a message is sent and then, at a later time, check the communications channel for messages.

The important difference between the two schemes is the need for buffering messages. In asynchronous communication, the sender may send many messages without the receiver removing them from the channel. Thus the channel must be prepared to buffer a potentially unlimited number of messages. If the number of messages in the channel is limited, eventually the sender will be blocked. In synchronous communication, only one message exists at any one time on the channel so no buffering is needed.

The difference is analogous to that between the telephone system and the postal system. A telephone call synchronizes the activities of the caller and the person who answers. Difficulties with synchronization cause busy signals, unanswered calls, etc. On the other hand, any number of letters may be dropped in a mail box at any time and the receiver may choose to check the incoming mail at any time. There is no synchronization.

Unbuffered synchronous communication is the lower-level concept and hence more efficient. It is the primitive of choice in the models of occam and Ada. If buffers are needed, they must be explicitly programmed on top of the synchronous system, just as answering machines are built to allow a caller to buffer a message with the receiver rather than block.

The problem with an asynchronous system is that we have to specify the buffering system at the time the distributed *primitives* are being designed rather than leaving it to the individual programmer. The language Linda which is based on asynchronous communication offers unparalleled expressibility and convenience at increased implementation cost.

7.3 Process Identification

A telephone company can install a line that directly connects two telephones. Such a dedicated line is usually of higher quality and speed than one that is

temporarily set up by the ordinary switching system. However, it is not flexible since it takes time to install and is very expensive.

Switching equipment allows any telephone to call any other. To call, the originator of the call must know the identification - the telephone number - of the destination, but the receiver cannot know the identification of the caller unless the caller explicitly chooses to pass on this information. Another way of saying this is that the caller may be anonymous, but not the receiver.

A more flexible means of communication is a bulletin board. Here both parties may choose to remain anonymous: an unsigned message may be posted and anyone is able to read or remove the message.

All these forms of naming are used in distributed models:

occam Dedicated channels connecting pairs of processes.

Ada A process calls another process by name without divulging its own identity.

Linda Broadcast messages (that need not be signed with a process identifier).

Dedicated channels are the most efficient since every message can be delivered with no overhead cost of deciphering addresses. However, any modification of the system entails changing the source code of the program.

The Ada asymmetric system is best suited to writing *server* processes, like a disk driver. Any process that knows the name of the server can request and receive the service that the process implements. The server need not be concerned with the identity of the requesters. Adding additional processes which need the same service to the system can thus be done without modification of source code.

Without process identifiers, the flexibility of the system is greatly improved since it is possible to add, remove or modify processes dynamically. For example, suppose that a system that used one type of disk drive was upgraded to use two different types. The new driver would have its own process name and we would have to change all requesting processes or at least write an additional process to route the requests. In Linda, one can simply add the needed driver and it will remove and process requests with no other change to the system.

7.4 Data Flow

A single act of communication can have data flowing in one direction or two. A telegram or letter causes data to flow in one direction only. A reply requires a separate act of communication. A telephone call allows two-way communication. While it is true that one party must call the other party by number, once the connection is made data can flow both ways.

Asynchronous systems (Linda) use one-way data flow since the sender can send the message and then continue. If the receiver is not ready, obviously it cannot return data to the sender and a separate message is needed.

In synchronous systems, a channel is set up between two processes and a decision must be made whether to implement one-way (occam) or two-way data flow (Ada). As usual, there is a tradeoff between expressibility and efficiency, though care is needed when comparing the two designs for efficiency. It is true

that sending a single message on a one-way channel is extremely efficient, because all that is needed is an acknowledgement of correct transmission and the sender need not be blocked while the receiver processes the message and decides if a reply is needed. However, if *most* messages do in fact need a reply, it may be more efficient to block the sender rather than release it and later go through the overhead of a new message for the reply.

7.5 Process Creation

Do all processes in a concurrent program exist when the program commences execution, or can processes be created dynamically? This question is not restricted to distributed systems, but it is convenient to discuss here since the three formalisms make different choices. As usual, a static definition of the set of processes (occam) is most efficient since the implementation can be done in terms of directly accessible compiled routines, memory locations, or channel numbers. There are several reasons that one would want the ability to create processes during execution.

Flexibility A system can be designed and programmed without knowing how many processes will be needed. During initialization, or upon request, processes can be allocated. For example, an operating system may allocate a process to each connected terminal. As the number of terminals changes, so can the number of allocated processes. On a static system, there would be a built-in limit to the number of connected terminals.

Dynamic use of resources A system may need different sets of resources at different times. With dynamic allocation, the number of processes for each resource can be matched with the requirements. This both saves the memory needed for process descriptors and also improves efficiency since implementations of concurrency may have algorithms that depend on the number of processes.

Load balancing In a multi-processor system, one possible design is to allocate a processor to each separate function. If one function temporarily requires increased computational power, there is little that can be done. However, if dynamic process creation is available, additional processes can be created to perform the overloaded function and assigned to underutilized processors. When the load on this function is reduced, the extra processes can be terminated and the processors allocated to other tasks.

Static process creation is most applicable in embedded systems like aircraft control systems or medical monitoring systems where the configuration of the system is fixed and predictability of performance is more important than flexibility. Since the configuration of the computer is also fixed, load balancing is not important. On a large transaction processing system, like an airline reservation system, dynamic process creation is important. The computing requirements change during the day and the system must not be halted during configuration changes needed to maintain the system or to keep it running during partial equipment failures.

7.6 Further Reading

A collection of articles that surveys the field of distributed programming can be found in [CDJ84]. Case studies of real systems can help motivate the study of distributed programming. Two that are relevant are [GS84] which describes an airline reservations system and [SG84] which describes the avionics system of a spacecraft.

Chapter 8

Ada

8.1 Introduction

Ada is a comprehensive programming language developed for the US Department of Defense as the standard language for critical software systems. Ada uses strong typing for reliability and a construct called *packages* for decomposition of large systems into modules. Our interest is in a third aspect of the language: the provision of built-in primitives for concurrent programming. Earlier languages supported only sequential programming; concurrent programming was done by directly calling the underlying operating system or by creating an operating system directly on the computer hardware. This made it practically impossible to port programs from one system to another. By including a model of concurrent programming, called *tasking* in the standard language, Ada makes this possible.

This chapter discusses Ada tasking in some detail. The model is very powerful and we can naturally express most concurrent algorithms in the language. We will also examine some of the shortcomings of Ada tasking. In the next two chapters, we will sketch the main ideas of two other concurrent programming models which took radically different decisions in the conflict between expressibility and efficiency.

Because Ada is an excellent and highly available language, it has been chosen as the unifying notation in this book. However, this is not a textbook on the language and simplifications or omissions may exist where a complete description would have caused too far a digression from our main subject. Several good texts on Ada are listed in the references for those who will actually use the language.

8.2 Rendezvous

Communication in Ada is synchronous (and unbuffered). Two tasks[1] must meet in a *rendezvous* in order to communicate. The name is chosen to invoke the image of two people who choose a place to meet (Figure 8.1(a)). The first one to arrive must wait for the arrival of the second.

[1] To conform with Ada terminology, processes will be called tasks in this chapter.

In this image, the two people are symmetric and the rendezvous is neutral. In Ada, however, the location of the rendezvous belongs to one of the tasks, called the *accepting* task. The other task, the *calling* task, must know the identity of the accepting task and the name of the location of the rendezvous (Figure 8.1(b)), called the *entry*. The accepting task, however, does not know the identity of the calling task. It is important to emphasize that both tasks are executing concurrently and they only synchronize at the rendezvous.

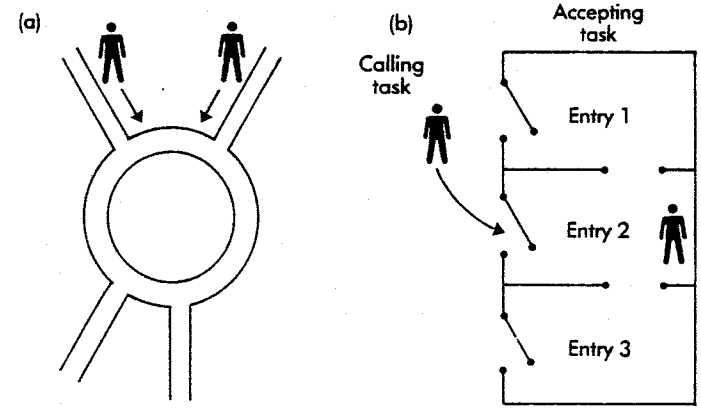

Figure 8.1 Rendezvous

The advantage of the asymmetric model is that *servers* can be naturally programmed. For example, the accepting task could be a printer server which will be willing to accept requests to print from any other task in the system. If a symmetric naming system were used, the programming of the server task would have to be changed every time a new task is added.

Let us turn to the definition of the rendezvous in Ada. A task is divided into two sections, the *specification* and the *body*. The specification may only contain declarations of *entries*.[2] Syntactically, an entry is exactly like a procedure declaration. This is done on purpose to allow substitution of a concurrent entry for a sequential procedure without otherwise changing a program.

```
task Buffer is
  entry Append(I: in  Integer);
  entry Take  (I: out Integer);
end Buffer;
```

Once a task specification has been declared and made visible to other tasks, they may call the entry using dotted notation – the task name followed by the entry name and appropriate parameters:

```
Buffer.Append(I);
```

[2] Until now, we have not used entries so we have omitted the task specification. However, every task must have a specification, even if it is null.

If Append were a procedure, control would be transferred immediately to the procedure body. An entry call must rendezvous with an accept statement in the task owning the entry:

```
task body Buffer is
begin
  ...
  accept Append(I: in Integer) do
    ... statements in the accept body
  end Append;
  ...
end Buffer;
```

Remember, the accepting task is an ordinary sequential process, executing its instructions. Like other instructions, the accept statement is executed in sequence when the instruction pointer reaches it, except that its definition requires synchronization with a calling task. In the assumed interleaving of execution sequences, either the calling task reaches the entry call before the accepting task reaches an accept statement for this entry, or conversely. Whichever task is first will suspend until the other reaches its matching statement. At that point, the rendezvous commences. The semantics of a rendezvous are as follows:

1. The calling task passes its in parameters to the accepting task and is blocked pending completion of the rendezvous.
2. The accepting task executes the statements in the accept *body* (the statements following the do up to the matching end).
3. The out parameters are passed back to the calling task.
4. The rendezvous is now complete and both tasks are no longer suspended.

Note that the completion of the rendezvous does not necessarily cause a task to commence execution. The decision of which task will execute is left to the underlying scheduler. The completion of a rendezvous completes the execution of the accept statement. If the task wishes to engage in an additional rendezvous, it must execute another accept statement. A typical non-terminating task, like a server, will have an accept statement in a loop so that it can engage in multiple rendezvous.

Figure 8.2 demonstrates the accept statement on a program that maintains a degenerate bounded buffer. Producer tasks will call the Append entry and consumer tasks will call the Take entry. The accept statements have been placed in a loop so that after each pair of rendezvous (between a producer and the buffer and then between a consumer and the buffer) the buffer task will re-issue the accept Append statement.

Comparing this solution with the monitor solution, we find that the buffer has become a separate task rather than just a passive set of data and procedures. However, even though there is the overhead of an additional task, on a multiprocessor implementation the Ada solution could be more efficient since not all of the processing has to be done within the mutual exclusion of the rendezvous. In this case, updating the indices and the counter need not be done in the rendezvous since they are local variables. This structure is typical of Ada algorithms since


```
task body buffer is
  B: array(0..N-1) of Integer;
  In_Ptr, Out_Ptr: Integer := 0;
  Count: Integer := 0;

begin
  loop
    accept Append(I: in Integer) do
      B(In_Ptr) := I;
    end Append;
    Count := Count + 1;
    In_Ptr := (In_Ptr + 1) mod N;
    accept Take(I: out Integer) do
      I := B(Out_Ptr);
    end Take;
    Count := Count - 1;
    Out_Ptr := (Out_Ptr + 1) mod N;
  end loop;
end Buffer;
```

Figure 8.2 Degenerate bounded buffer

synchronization is defined directly between tasks and not through the agency of an operating system.

The entries belong to the task which declares them and only it can execute accept statements for them. While the accepting task can have many accept statements (even for the same entry), it can obviously only execute one statement at a time. However, several tasks can call the same entry in another task. For example, several producers could call Buffer.Append concurrently. Tasks calling the same entry are enqueued in order of arrival on the entry. Each accept statement engages in a rendezvous with the first task in its entry queue. Completion of the accept body completes the rendezvous; the accepting task does *not* continue to engage in rendezvous with the tasks in the entry queue. Instead, it must execute another accept for this entry (i.e. another occurrence of an accept statement, which may be the same statement encountered after looping).

To summarize, the Ada rendezvous is a primitive with the following characteristics:

- Synchronous, unbuffered communication.
- Asymmetric identification – the sender knows the identity of the receiver, but not conversely.
- Two-way data flow during a single rendezvous.

Global variables that are shared among several tasks are also allowed in Ada but we refer the reader to an Ada textbook for a description of this feature. They have been used to program the examples in Chapter 3 in Ada.