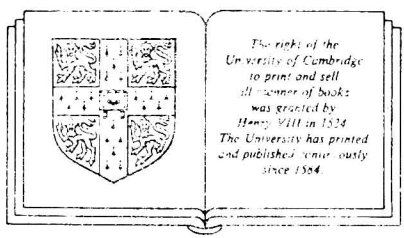
26 Cambridge Computer Science Texts

Concurrent Programming

26 Cambridge Computer Science Texts

Concurrent Programming

C. R. Snow
University of Newcastle upon Tyne



Cambridge University Press

Cambridge
New York Port Chester Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Victoria 3166, Australia

First published 1992

Printed in Great Britain at the University Press, Cambridge

Library of Congress cataloguing in publication data available

British Library cataloguing in publication data available

ISBN 0 521 32796 2 hardback
ISBN 0 521 33993 6 paperback

Also in this series

1 An Introduction to Logical Design of Digital Circuits
C. M. Reeves 1972

2 Information Representation and Manipulation in a Computer
E. S. Page and L. B. Wilson, Second Edition 1978

3 Computer Simulation of Continuous Systems
R. J. Ord-Smith and J. Stephenson 1975

4 Macro Processors
A. J. Cole, Second Edition 1981

5 An Introduction to the Uses of Computers
Murray Laver 1976

6 Computing Systems Hardware
M. Wells 1976

7 An Introduction to the Study of Programming Languages
D. W. Barron 1977

8 ALGOL 68 – A first and second course
A. D. McGettrick 1978

9 An Introduction to Computational Combinatorics
E. S. Page and L. B. Wilson 1979

10 Computers and Social Change
Murray Laver 1980

11 The Definition of Programming Languages
A. D. McGettrick 1980

12 Programming via Pascal
J. S. Rohl and H. J. Barrett 1980

13 Program Verification using Ada
A. D. McGettrick 1982

14 Simulation Techniques for Discrete Event Systems
I. Mitrani 1982

15 Information Representation and Manipulation using Pascal
E. S. Page and L. B. Wilson 1983

16 Writing Pascal Programs
J. S. Rohl 1983

17 An Introduction to APL
S. Pommier 1983

18 Computer Mathematics
D. J. Cooke and H. E. Bez 1984

19 Recursion via Pascal
J. S. Rohl 1984

20 Text Processing
A. Colin Day 1984

21 Introduction to Computer Systems
Brian Molinari 1985

22 Program Construction
R. G. Stone and D. J. Cocke 1987

23 A Practical Introduction to Denotational Semantics
Lloyd Allison 1987

24 Modelling of Computer and Communication Systems
I. Mitrani 1987

25 The principles of Computer Networking
D. Russel 1989.

Preface

For a number of years, Concurrent Programming was considered only to arise as a component in the study of Operating Systems. To some extent this attitude is understandable, in that matters concerning the scheduling of concurrent activity on a limited number (often one!) of processing units, and the detection/prevention of deadlock, are still regarded as the responsibility of an operating system. Historically, it is also the case that concurrent activity within a computing system was provided exclusively by the operating system for its own purposes, such as supporting multiple concurrent users.

It has become clear in recent years, however, that concurrent programming is a subject of study in its own right, primarily because it is now recognised that the use of parallelism can be beneficial as much to the applications programmer as it is to the systems programmer. It is also now clear that the principles governing the design, and the techniques employed in the implementation of concurrent programs belong more to the study of programming than to the management of the resources of a computer system.

This book is based on a course of lectures given over a number of years initially to third, and more recently to second year undergraduates in Computing Science. True to the origins of the subject, the course began as the first part of a course in operating systems, but was later separated off and has now become part of a course in advanced programming techniques.

We make the assumption that the reader is very familiar with a high-level sequential programming language such as Pascal, and that the examples given throughout the book can be read by a reasonably competent programmer. Certainly, the students to whom the course is given have received a thorough grounding in Pascal programming prior to taking the course.

Sadly, it is still the case that there is very little uniformity about the facilities available (if any) to students wishing to run concurrent programs. It has therefore been rather difficult to provide good programming exercises to accompany the text. Ideally, an environment will be available in which a number of styles of concurrent programming can be tried and compared, but even now such circumstances are rarely available to the average undergraduate class. Encouraging signs are to be seen, however, and concurrency facilities are likely to be available in the most of the next generation of general purpose programming languages which will be candidates for use as first programming languages for student teaching.

Where a system is available which offers facilities for concurrent programming, particularly one which supports the shared memory model, a programming exercise has been suggested (in chapter 3) which may be used to demonstrate the phenomenon of interference (also discussed in chapter 3). A common difficulty in locating bugs in concurrent programs is their reluctance to manifest themselves. It is hoped that this exercise will help to demonstrate that the problem of interference does exist, and that the proposed solutions really do reduce or eliminate the problem.

On the subject of practical work in general, many of the exercises have been constructed in such a way as to suggest that a concurrent program should be written. It is strongly recommended that, if the necessary facilities are available, the exercise solutions should be coded and run, in order to reinforce the points made, and the techniques described in the text.

I have resisted the temptation to include a comprehensive bibliography and have given instead a fairly brief set of references, suitably annotated, which are intended to complement the specific topics covered in the book. A number of these references themselves contain bibliographies of the wider literature on the subject, to which the interested reader is invited to refer.

It is with great pleasure that I acknowledge my gratitude to all those friends and colleagues who have encouraged me through the lengthy gestation period of this book. It is surprising how the simple question "How's the book coming along?" can be a spur to the author. I am also happy to record my thanks to my faithful Xerox 6085 workstation "Brinkburn" running the Viewpoint software, upon whic

the whole book has been typeset. Finally, this preface would not be complete without recording my enormous appreciation of and thanks to the numberless students who have acted as the sounding board for the views expressed herein. Their reactions to the course, their questions, and to some extent their answers to my questions, have influenced this book in more ways than they, or indeed I, could have realised.

C.R.Snow

Newcastle upon Tyne, June 1991.

Contents

1
Introduction to Concurrency

Concurrency has been with us for a long time. The idea of different tasks being carried out at the same time, in order to achieve a particular end result more quickly, has been with us from time immemorial. Sometimes the tasks may be regarded as independent of one another. Two gardeners, one planting potatoes and the other cutting the lawn (provided the potatoes are not to be planted on the lawn!) will complete the two tasks in the time it takes to do just one of them. Sometimes the tasks are dependent upon each other, as in a team activity such as is found in a well-run hospital operating theatre. Here, each member of the team has to co-operate fully with the other members, but each member has his/her own well-defined task to carry out.

Concurrency has also been present in computers for almost as long as computers themselves have existed. Early on in the development of the electronic digital computer it was realised that there was an enormous discrepancy in the speeds of operation of electro-mechanical peripheral devices and the purely electronic central processing unit. The logical resolution of this discrepancy was to allow the peripheral device to operate independently of the central processor, making it feasible for the processor to make productive use of the time that the peripheral device is operating, rather than have to wait until a slow operation has been completed. Over the years, of course, this separation of tasks between different pieces of hardware has been refined to the point where peripherals are sometimes controlled by a dedicated processor which can have the same degree of "intelligence" as the central processor itself.

Even in the case of the two gardeners, where the task that each gardener was given could be considered to be independent of the other, there must be some way in which the two tasks may be initiated. We

may imagine that both of the gardeners were originally given their respective tasks by the head gardener who, in consultation with the owner of the garden, determines which tasks need to be done, and who allocates tasks to his under-gardeners. Presumably also, each gardener will report back to the head gardener when he has finished his task, or maybe the head gardener has to enquire continually of his underlings whether they have finished their assigned tasks.

Suppose, however, that our two gardeners were asked to carry out tasks, both of which required the use of the same implement. We could imagine that the lawn-mowing gardener requires a rake to clear some debris from the lawn prior to mowing it, while the potato planter also requires a rake to prepare the potato bed before planting. If the household possessed only a single rake, then one or other gardener might have to wait until the other had finished using it before being able to complete his own task.

This analogy serves to illustrate the ways in which peripheral devices may interact with central processors in computers. Clearly if we are asking for a simple operation to take place, such as a line printer skipping to the top of the next page, or a magnetic tape rewinding, it suffices for the Central Processing Unit (c.p.u.) to initiate the operation and then get on with its own work until the time when either the peripheral device informs the c.p.u. that it has finished (i.e. by an interrupt), or the c.p.u. discovers by (possibly repeated) inspection that the operation is complete (i.e. by polling). Alternatively, the peripheral device may have been asked to read a value from an external medium and place it in a particular memory location. At the same time the processor, which is proceeding in its own time with its own task, may also wish to access the same memory location. Under these circumstances, we would hope that one of the operations would be delayed until the memory location was no longer being used by the other.

1.1 Reasons for Concurrency

Concurrent programming as a discipline has been stimulated primarily by two developments. The first is the concurrency which had been introduced in the hardware, and concurrent programming could be seen as an attempt to generalise the notion of tasks being allowed to proceed largely independently of each other, in order to mimic the

relationship between the various hardware components. In particular, the control of a specific hardware component is often a complex task requiring considerable ingenuity on the part of the programmer to produce a software driver for that component. If a way could be found by which those aspects of the driver which are concerned with the concurrent activity of the device might be separated off from other parts in the system, the task is eased tremendously. If concurrent programming is employed, then the programmer can concern himself with the sequential aspects of the device driver, and only later must he face the problem of the interactions of the driver with other components within the system. In addition, any such interactions will be handled in a uniform and (hopefully) well-understood way, so that (device-) specific concurrency problems are avoided.

The second development which leads directly to a consideration of the use of concurrent programming is a rationalisation and extension of the desire to provide an operating system which would allow more than one user to make use of a particular computer at a time. Early time-sharing systems which permitted the simultaneous use of a computer by a number of users often had no means whereby those users (or their programs) could communicate with one another. Any communication which was possible was done at the operating system kernel level, and this was usually a single monolithic program which halted all the user tasks while it was active. Users of such systems were not generally concerned with communicating with each other, and the only form of resource sharing that they required was in the form of competition for resources owned by the operating system. Later systems which came along began to require the possibility of users sharing information amongst themselves, where such information was not necessarily under the control of the operating system. Data could frequently be passed from one user program to another much more conveniently than using a cumbersome mechanism of asking one program to write data into a file to be read by the other program.

The introduction of concurrent programming techniques was also recognised to be a useful tool in providing additional structure to a program. We remarked earlier that the task of constructing a device driver is considerably simplified if the concurrency aspects can be set aside, and then added in a controlled way. This is very similar in

concept to some of the well-established techniques of structured programming, in which the communication between the various parts of a (sequential) program is strictly controlled, for example through the use of procedures and parameter lists. Structured programming also leaves open the possiblity of delaying the coding of various parts of the program until a later, more convenient time, allowing the writer of the program to concentrate on the specific task on hand.

In a similar way, the writer of a concurrent program may write a sequential program, leaving aside the questions of the interaction with other concurrently active components until the sequential program is complete and, possibly, partially tested. We suspect that unstructured parallelism in programming would be even more difficult to manage than an unstructured sequential program unless we were able to break down the concurrency into manageable sub-units. Concurrent programming may therefore be regarded as another manifestation of the "divide and conquer" rule of program construction. Such methodologies are also useful as a way of making programs more readable and therefore more maintainable.

1.2 Examples of Concurrency

There are many useful examples of concurrency in everyday life, in addition to the example of the two gardeners mentioned above. Any large project, such as the building of a house, will require some work to go on in parallel with other work. In principle, a project like building a house does not require any concurrent activity, but it is a desirable feature of such a project in that the whole task can be completed in a shorter time by allowing various sub-tasks to be carried out concurrently. There is no reason why the painter cannot paint the outside of the house (weather permitting!), while the plasterer is busy in the upstairs rooms, and the joiner is fitting the kitchen units downstairs. There are however some constraints on the concurrency which is possible. The bricklayer would normally have to wait until the foundations of the house had been laid before he could begin the task of building the walls. The various tasks involved in such a project can usually be regarded as independent of one another, but the scheduling of the tasks is constrained by notions of "task *A* must be completed before task *B* can begin".

A second example is that of a railway network. A number of trains may be making journeys within the network, and by contrast with the previous example, when they start and when they end is generally independent of most of the other journeys. Where the journeys do interact though, is at places where routes cross, or use common sections of track for parts of the journeys. We can in this example regard the movement of the trains as programs in execution, and sections of track as the resources which these programs may or may not have to share with other programs.

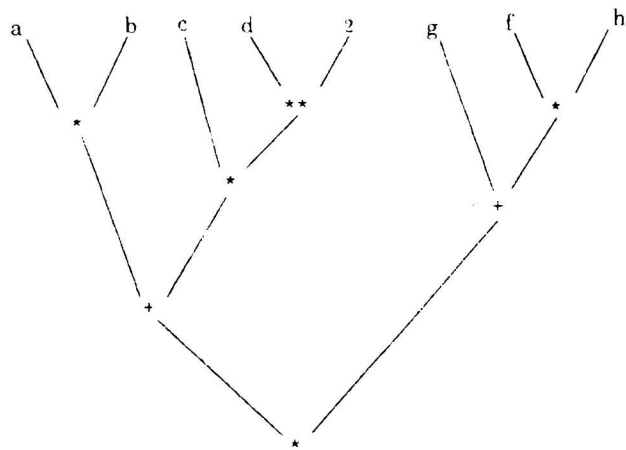
In some cases, the concurrency is inherent in the situation being considered. Any complex machine, or large plant such as a power station, chemical works or oil refinery, consists of identifiable components which have to be continuously interacting with other components. In a quality-controlled environment, for example, the product of the manufacturing component of the system is subjected to certain test procedures, which in turn provide information which may modify the way in which the manufacturing component behaves. Clearly these two components need to be constantly active and in constant communication with each other for the whole system to work properly.

An example of concurrency directly related to computing and programming can be seen by considering the evaluation of an arithmetic expression. Suppose we wish to evaluate the expression:

$$(a*b + c*d**2)*(g + f*h)$$

We assume that the identifiers a, b, c, etc. have values associated with them, and that the priority rules for evaluation of the expression are as would be expected, i.e. exponentiation first, multiplication second and addition last, modified in the usual way by the inclusion of parentheses. A tree may be drawn (figure 1.1) showing the interdependencies of the sub-expressions within the whole expression, and we may use this tree to identify possible concurrency within the evaluation. Three concurrent evaluations of sub-expressions can begin at once, namely, $a*b$, $d**2$ and $f*h$. When the second and third of these are finished, the multiplication by c, and the addition of g (respectively) can take place, also in parallel. It is only after $c*d**2$

Figure 1.1



has been evaluated that the sub-expression $a*b$ can be added, and then finally the evaluation of the whole expression can be completed.

It is in the field of operating systems where concurrent programming has been most fruitfully employed. A time-sharing operating system, by its very nature, is required to manage several different tasks in parallel, but even if the system is only providing services to a single user it will be responsible for managing all the peripheral devices as well as servicing the user(s). If the concurrent programming facilities can also be offered to the user, then the flexibility of concurrency as a program structuring technique is also available to application programs. It is not our intention in this book to deal with operating systems as a subject, but it will inevitably be the case that operating systems will provide a fertile source of examples of concurrent programs.

1.3 Concurrency in Programs

Any sequential program is in all probability not necessarily totally sequential. That is, it is often the case that the statements of the program could be re-ordered to some extent without affecting the behaviour of the program. It is usually possible, however, to identify those parts of the program (it is useful to think in terms of program statements in your favourite high-level programming language) which do depend on one another, in the sense that one statement must

precede another. A simple example would be the case where a program might include the statement

$$x := x + 1;$$

This statement should really only be used if there has been a previous statement initialising the variable x, e.g.

$$x := 0;$$

We could examine any sequential program and identify all such dependencies, and we would almost certainly find that there were quite a number of pairs of statements for which no such dependencies exist. In other words, it is possible to identify a partial ordering of program statements which define the interdependencies within a program.

Suppose, for example, that a program consisted of statements denoted by the letters $A, B, C, D, E, F, G, H, I$ and J. Suppose also that we were able to write down the partial ordering as a set of relations:

$$A < B, A < C, A < D, C < E, D < E, B < F,$$
$$D < F, E < F, F < G, F < H, G < I, H < J$$

where the relational operator $<$ is meant to be interpreted as "must precede in time". The first property of this partial ordering we notice is that the relation $D < F$ is in fact unnecessary, since it is a consequence of the two relations $D < E$ and $E < F$, this ordering relation having the transitivity property. The partial ordering defined by these relations may be illustrated by a directed graph as shown in figure 1.2.

1.4 An Informal Definition of a Process

It is assumed that any programmer who attempts to write concurrent programs will have had a reasonable amount of experience of conventional sequential programming. With this in mind, we attempt to decompose our concurrent programming problem into a set of sequential programs together with some controlled interaction between them. Thus we put forward as the basic building block of a concurrent program the *sequential process* (where no confusion can