Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

95

Christopher D. Marlin

Coroutines

A Programming Methodology, a Language Design and an Implementation

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

95

Christopher D. Marlin

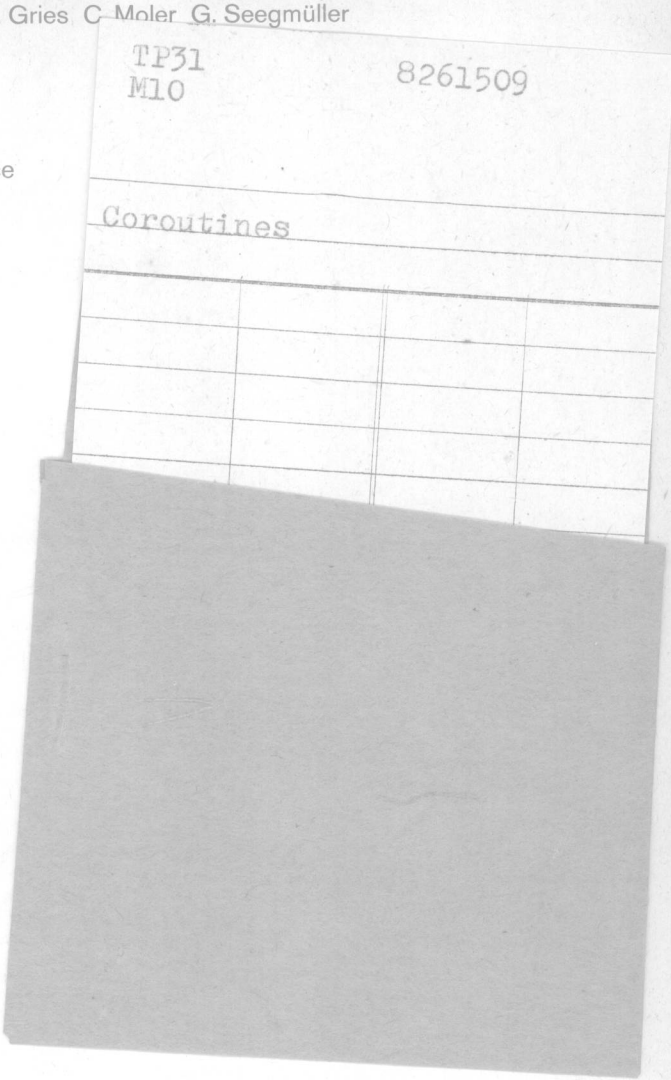
Coroutines

A Programming Methodology, a Language Design and an Implementation

Springer-Verlag
Berlin Heidelberg New York 1980

Author

Christopher D. Marlin
Department of Computer Science
101 MacLean Hall
The University of Iowa
Iowa City, Iowa 52242/USA



AMS Subject Classifications (1980): 68-02, 68B05, 68F20
CR Subject Classifications (1974): 4.0, 4.12, 4.20, 4.22

ISBN 3-540-10256-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10256-6 Springer-Verlag New York Heidelberg Berlin

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

PREFACE

Coroutines have been known and discussed for some years, but unfortunately have acquired a reputation for leading to poorly-structured and inefficient programs. It is perhaps a consequence of this unjustified reputation that coroutines are not widely available in programming languages.

The work described in this volume began both as an investigation of methodologies for programming with coroutines and as an attempt to extend the notion of hierarchical program structure to programs involving coroutines. The results of these efforts are presented in Chapter 2.

Inadequate support for hierarchically-structured systems of coroutines in existing languages then motivated the design of a language with coroutines. Although they are not widely available in implemented programming languages, coroutines have been described and discussed extensively in the literature, with a large number of proposals for the inclusion of coroutines in programming languages being put forward. The approach to language design described in Chapter 3 was born out of a desire to draw on the experience represented by this body of coroutine-related literature. This approach involves:

- the design of semantics before that of syntax,
- the division of the design of the semantics of a language into that of three largely orthogonal aspects of the language (data structures, sequence control, and data control), and
- the use of specific abstract models to aid the design of the semantics of each of these aspects, by facilitating comparisons among previous languages and proposals, and among competing design options for the language being designed.

The result of applying this approach to the design of a language with coroutines (known as ACL) is described in Chapters 4 (semantics) and 5 (syntax). This language was designed with relatively efficient implementation as one of its goals, and Chapter 6 describes some aspects of an implementation which has been carried out.

Apart from some minor corrections and editorial changes, this volume reproduces a thesis submitted by the author to the University of

Adelaide, Adelaide, South Australia, for the degree of Doctor of Philosophy, on 16th November 1979.

I gratefully acknowledge the support and encouragement of my supervisor, Dr.C.J.Barter. My thanks are also due to Dr.J.G.Sanderson for many helpful discussions, particularly while acting temporarily as my supervisor, and to the many other people who assisted me while I was carrying out the work described here. I would also like to thank Prof. D.L.Epley of the University of Iowa for his thoughtful advice and comments.

Above all, I am grateful to my wife, Deborah, for her constant support and for her unfailing confidence in my ability to finish the task I had set myself; she is also responsible for the careful preparation of the many diagrams in this volume.

Iowa City, Iowa C.D.M.
July 1980

TABLE OF CONTENTS

Chapter Page

Chapter Page

Chapter

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Coroutines

The invention of the word "coroutine" is attributed to Conway[29] who describes a coroutine as "an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program".

This view of coroutines as "mutual subroutines"[43] has remained the most common view of how coroutines can be used; it is epitomized by the classic example of a parser calling on a syntax analyser for the next token, and that lexical analyser calling on the parser to dispose of a token just extracted from the input sequence. There is also a relationship, discussed by Knuth[85], between multi-pass algorithms and coroutines, which allows multi-pass algorithms to be implemented using coroutines, in such a way that the execution of the passes is interleaved.

For the purposes of this thesis, the following will be regarded as the fundamental characteristics of a coroutine:

(1) the values of data local to a coroutine persist between successive occasions on which control enters it (that is, between successive calls), and

(2) the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

These characteristics describe a mechanism which allows coroutines to call each other in a symmetric fashion, and to pass control back and forth between each other. Characteristics such as these have lead to the

view that coroutines are some special kind of procedures with "own variables" (see, for example, Fisher's discussion in [43]); the contrary view is expressed later in this chapter that procedures are a special form of coroutines.

Within the constraints represented by the above characteristics, there is still some flexibility as to the manner in which the coroutines execute:

(a) If the coroutines communicate only via first-in-first-out (FIFO) queues, and if there is no explicit transfer of control between the coroutines, then all inter-coroutine interactions can be regarded as interactions between a producer (a coroutine writing an item onto a queue) and a consumer (a coroutine which reads the item from the queue). Under these conditions, there arises the possibility of parallel (either virtual or actual) execution of the coroutines, when several of the coroutines are not waiting on any of their producers.

(b) Alternatively, it is possible to transfer control explicitly from one coroutine to another, causing the currently executing coroutine to become suspended and a target coroutine to resume execution. In this case, only one coroutine is ever executing at any given time, and there arises no question of parallel execution.

The first of these situations will be referred to as the "implicit sequencing" kind of coroutine and it can be argued, as does Constantine[28], that this was the kind of coroutine described by Conway; Constantine goes so far as to say that Conway's coroutines are addressed precisely to the elimination of explicitly specified sequencing. A good example of the implicit sequencing kind of coroutine is provided by the system of Kahn and MacQueen[82], in which the coroutines form networks and communicate only by reading from, and writing onto, one-way communication channels (queues); Kahn and MacQueen also point out the relationship between this kind of coroutine and "call-by-need" parameters, "lazy evaluation" and "streams". Dennis[34] describes a variant of the implicit sequencing kind of coroutines, which he calls a "data flow representation", in which the inter-coroutine communication queues are limited to a maximum length of one, and are

called "communication variables"; writing a value into a communication variable is held up until any previous value has been consumed by all modules for which the variable provides values, and reading a value from a communication variable is held up until any previously read value is replaced by a new one.

In all its forms, the implicit sequencing kind of coroutine represents a language feature in which the flow of control is not explicitly specified, but is dynamically determined by data dependencies in the program (that is, by the "data flow"); the execution of such a program can be said to be "data-driven".

The second kind of coroutine described above, the "explicit sequencing" kind, is the one which is well-known to assembly language programmers (for example, see [85,144]), although it seldom receives a great deal of attention in the context of higher level languages. It is this kind of coroutine which was the sole subject of the investigation described here, and will be discussed exclusively in the remainder of this thesis; the choice of the explicit sequencing kind of coroutine in this case is not intended as a statement of preference for it over the implicit sequencing kind. It is, however, the author's view that the two kinds of coroutine should be distinguished and their study separated. Some comments are made at the end of this thesis as to how the ideas presented herein might be extended to the implicit sequencing kind of coroutine.

It is possible to regard the explicit sequencing kind of coroutine as a restricted form of the implicit sequencing kind, as is done by Grune[59], for example; it is also possible to view the explicit sequencing kind of coroutine as a means of implementing or formulating the implicit sequencing kind (see, for example, [82,114]).

Probably the earliest published form of coroutines were the "generators" of Information Processing Language V (IPL-V), a symbol and list structure manipulation language described by Newell and Tonge[113]; the purpose of IPL-V's generators is the production of a sequence of outputs, and the application of a specified process to each of the outputs, to achieve the effect of an iteration statement. One process, called the "superprocess", calls the generator and passes to it the process (called the "subprocess") to be applied to the outputs, as an

argument. The generator then produces the first output and calls the subprocess; after applying itself to an output, the subprocess calls the generator, signalling either that it requires another element from the sequence, or that no further values are required. In the former case the generator will return control to the subprocess after the production of the new value, and in the latter case it will return control to the superprocess, its task complete. The subprocess and superprocess execute in one context, and the generator in another; thus, an amount of context switching attends the transfers between generator and subprocess. During the phase that the generator is producing values to which the subprocess applies itself, the former can be regarded as subordinate to the latter and the relationship between them can be regarded as asymmetric, being that between a calling and a called routine. The fact that the generator executes in a different context from its caller, and that this context is saved between successive calls, gives the generator a coroutine-like behaviour. The relationship between the generator and the superprocess, on the other hand, is more like that between a procedure and its caller, since the generator restarts the production of values from the sequence each time that it is called by the superprocess.

From these origins, the term "generator" has come to be used for a coroutine whose behaviour is restricted in that it returns to its caller (invoker) on completion of its task. However, a generator is still a coroutine, in the sense that the values of local data are retained between calls, and execution of a generator always continues from where it left off when it returned from the previous call. The "semicoroutine" facility of the Simula language[32] and the "semi-symmetric coroutine linkage" described by Wang and Dahl[152] both provide a similar capability to generators.

A procedure can be regarded as a form of coroutine which has further restrictions on its behaviour, in addition to those for a generator: it is a generator which starts with a fresh set of local data and commences execution from its first statement (instruction) on each occasion that it is called, but becomes suspended (in the coroutine manner) if it calls another subprogram, having its execution resumed at the point of that call with the same local data values when that procedure returns.