SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 765–771 (1980)

A Coroutine Mechanism for BCPL

KEN MOODY AND MARTIN RICHARDS
The Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge, U.K.

SUMMARY

In recent years interest in coroutines has increased considerably and their usefulness is no longer in question. This paper describes a simple coroutine mechanism that fits neatly into the philosophy of the BCPL language. A few diverse applications of the mechanism are presented at the end.

KEY WORDS Coroutine BCPL TRIPOS

INTRODUCTION

BCPL is a portable systems programming language that is well suited to a wide variety of applications including the implementation of compilers and operating systems. It is described in Richards.[1,2] Its most notable features are that all values in the language are of the same size and that there is no type checking either by the compiler or at run time. There are several languages that are similar or related to BCPL, for example C,[3] Eh[4] and BLISS.[5]

BCPL is a block-structured language with a recursive procedure calling mechanism. It uses a stack at run time for the storage of arguments, local variables and anonymous results, and there is a static area called the global vector which holds non-local variables, particularly those shared between separately compiled segments of program. Occasionally, applications arise where it is convenient to have a number of independent run time stacks, and such facilities can be provided by a simple implementation of coroutines.

In recent years there has been much interest in coroutines and several papers have appeared on the subject, for example Grune,[6] Jacobsen[7] and Gentleman.[8] However, there are still few languages having built-in coroutine facilities. One reason for this is that it is hard to provide a clean coroutine extension to an existing strictly typed language. Further, such an extension may require a substantial modification of the compiler, causing the cost of its implementation to outweigh the value of the new facilities provided. These disadvantages do not arise when augmenting a typeless language such as BCPL, since type issues are not relevant and the new facilities can be provided entirely by means of library procedures without the need for any compiler modifications.

THE BCPL COROUTINE MECHANISM

In this mechanism all coroutines share the same global vector but have distinct run time stacks, and it is thus natural to represent a coroutine by a pointer to its stack area. The

0038–0644/80/1010–0765$01.00 *Received 28 April 1980*

execution of a collection of coroutines behaves like a single sequential process, with just one coroutine (the *current* coroutine) having control at any particular moment. At that time all the other coroutines are said to be suspended. Every suspended coroutine has a *resumption point* from which execution continues when it next receives control. In order to transfer control to another coroutine, the current coroutine must invoke one of the library functions Callco, Cowait or Resumeco. Each of these will suspend the current coroutine, storing the resumption point in a standard place near the base of its stack before transferring control. When control next returns to this coroutine, it will be as if the call of Callco, Cowait or Resumeco has just yielded a result. Whenever control passes from one coroutine to another a value is also passed.

A typical use of the function Callco is given in the following assignment statement:

```
Val := Callco(Cptr, Arg)
```

Here, Cptr holds the identity of the coroutine to be called and Arg is the value to be passed. The calling coroutine becomes the *parent* of the called coroutine, and its identity is stored at a standard place in the called coroutine's stack. If the called coroutine already has a parent then the call is invalid. When control returns to the calling coroutine the value that is passed becomes the result of Callco, which, in this example, is assigned to Val.

Cowait is (normally) used in conjunction with Callco and provides a simple way of transferring control back to the parent. A typical example of its use is the following assignment:

```
Val := Cowait(Res)
```

Here, the current coroutine suspends itself and returns control to its parent, passing the value Res. When it next receives control the function application Cowait(Res) will complete, causing the value to be assigned to Val. A coroutine suspended by Cowait is left without a parent.

A scheme involving Callco and Cowait only would have a wide range of applications, but Resumeco is so useful that it has been included as one of the primitives. The following assignment contains a typical call of Resumeco:

```
Val := Resumeco(Cptr, Arg)
```

The effect of Resumeco is almost identical to that of Callco, differing only in its treatment of the parent. With Resumeco the parent of the calling coroutine becomes the parent of the called coroutine, leaving the calling coroutine suspended and without a parent. Systematic use of Resumeco reduces the number of coroutines having parents and hence allows greater freedom in organizing the flow of control between coroutines. Applications of Resumeco are given later. A special case is when Resumeco is called upon to resume the current coroutine. Although it is never valid for Callco to transfer control to a coroutine that already has a parent, it is permitted for Resumeco in this special case. Thus if Cptr identifies the current coroutine, the function application

```
Resumeco(Cptr, Arg)
```

immediately yields Arg as its result.

If a BCPL program is organized as a set of coroutines there must be one (called the root) which is the first to gain control. The root coroutine has a parent which is special in that control cannot be transferred to it by means of Cowait, nor may it become the parent of another coroutine by means of Resumeco.

To summarize, each coroutine must be either (1) the current coroutine, in which case it has a parent, or (2) suspended with a parent, or (3) suspended without a parent. There is a chain through the parent link from the current coroutine to the root coroutine, and all coroutines not in this chain are suspended without parent. The chain behaves like a stack with Callco increasing its length by one, Cowait decreasing it by one and Resumeco leaving the length unchanged.

CREATION AND DELETION

Coroutines can be created and deleted dynamically using the library procedures Createco and Deleteco. A typical call of Createco is given in the following assignment:

```
Cptr : = Createco(F, Size)
```

Here, F is the main procedure of the coroutine and Size specifies the amount of stack space that F requires. If sufficient space is available, Createco allocates and initializes the coroutine stack returning a pointer to it as result. The newly created coroutine is initially in a suspended state just as if Cowait had been called in the following repeated command:

```
C : = F(Cowait(C)) REPEAT
```

When the coroutine is activated, the value passed thus becomes the argument of F. If Cowait is called during the execution of F the coroutine suspends in the normal way; however, if F returns its result is assigned to C and immediately becomes the argument of Cowait. This causes the coroutine to suspend itself, passing the result of F back to the parent and leaving the coroutine again in its initial state.

Deleteco takes a coroutine as its argument and, after checking that it has no parent, returns its stack to free store.

RECURSIVE COROUTINES

Cowait and Resumeco leave the current coroutine suspended without a parent, whereas Callco leaves it suspended but with a parent. Since any suspended coroutine has a well-defined resumption point it is natural to consider whether it is ever sensible to reactivate (by Callco or Resumeco) a coroutine suspended by Callco. Such an activation is explicitly (and intentionally) prohibited in the mechanism described above, but is not an entirely ludicrous facility and could be implemented without difficulty by means of a stack of active coroutines in place of the parent chain. This generalized scheme was rejected on the grounds that it was difficult to find real situations where the extra power it provided was really necessary. Indeed, it was felt that if such a situation were found the resulting program would probably be rather obscure. There would also be the problem of deciding how large the coroutine activation stack should be.

The idea of recursive coroutines have been explored before, see, for instance, Krieg.[9]

EFFICIENCY

The efficiency of Createco and Deleteco is dependent mainly on the efficiency of the free store allocation routines, which can vary considerably from one implementation to another. However, the efficiencies of Callco, Resumeco and Cowait are easier to

measure and are more consistent. Each of these functions causes control to pass from one coroutine to another, and the number of machine instructions executed to effect such a transfer is a good measure of its efficiency. On the IBM 370, 15 instructions are executed in the body of Cowait before the instruction at the resumption point in the parent coroutine is obeyed. Table I below gives the number of instructions executed in Callco, Resumeco and Cowait for various machines.

Table I

Machine	Callco	Resumeco	Cowait
IBM 370	17	20	15
CAP	12	16	10
PDP 11	15	18	11
NOVA	19	23	17
LSI-4	19	27	18

The CAP computer is a 32-bit word addressed machine built at Cambridge for research into memory protection techniques and is described in Wilkes and Needham.[10]. In the IBM implementation more attention was paid to run time error diagnosis than in the others, but even so its efficiency is still adequate.

APPLICATIONS

There are several applications in which this coroutine mechanism has been used to good effect, and some of them are now described. The first two examples are taken from the TRIPOS operating system (see Richards[11]).

TRIPOS is a portable operating system implemented mainly in BCPL for use on medium sized mini- or micro-computers. It is designed primarily as a single user system offering real time facilities. The system is structered into independent BCPL tasks, each having its own stack and global vector. The BCPL global vector is a region of store analogous to blank COMMON in FORTRAN and is used mainly for communication between separately compiled modules. TRIPOS tasks communicate with each other using procedures defined in the TRIPOS kernel.

The command language interpreter (CLI) is a task in the TRIPOS system, and its job is to read and execute TRIPOS commands. For each command the CLI loads the appropriate module and then transfers control to its entry point. The CLI uses a known amount of stack space, but the loaded command may use any amount. To allow for this, a coroutine with an appropriate stack size is created with the loaded command as its body. Control is then passed to it by means of Callco. When the command has finished execution it returns control to the CLI, which deletes the coroutine and its code before processing the next command. The coroutine mechanism is used here solely to allow for the allocation of a dynamically computed amount of stack space.

A more significant use of coroutines occurs in the TRIPOS file handler task. This task provides filing system facilities to user tasks while simultaneously driving the disc device. For much of the time, the file handler task is in a state of voluntary suspension waiting for a message to arrive from either the disc device or a user task. The file handler processes such events in the order in which they occur. Some events, such as a request to open a file, may involve real time delays resulting from the need to access the disc. Such events are conveniently serviced by means of a sequential process with voluntary

halt points. A coroutine activation behaves in precisely this way, with the voluntary suspension being effected by Cowait. In the file handler there are three coroutines: ROOT, WORK and ACCESS. Messages to the file handler are received by ROOT which passes disc events to ACCESS and schedules user requests through WORK. Note that there is no need for an additional interlock mechanism to control access to data structures shared by the file handler processes, since each coroutine is guaranteed exclusive access until it suspends itself.

Another significant use of this coroutine mechanism is in the CODD Relational Database Management System implemented by T. J. King and described in King and Moody.[12] This system uses the pipeline technique pioneered in PRTV (Todd[13]). In PRTV a relational expression is evaluated by associating procedure activations with the nodes of the parse tree and then calling the resulting structure from the root. Index requests are transmitted from the root to the leaves, with the data tuples flowing in the opposite direction. Information local to each node determines the action to be taken when the local procedure is invoked.

In CODD a coroutine is created to correspond to each node. Local status is recorded both by the saved resumption point and by the variables in the local stack. The use of coroutines enables operations such as protocol sequences to be programmed conveniently. The procedures that implement relational operations in this way are easy to understand.

The use of coroutines has led to an evaluation model that is more general than that of PRTV. COPY nodes that split pipes have been introduced and are useful if, for example, an evaluated relational expression is to be both saved and listed. The resulting node structure can thus be any acyclic directed graph. In addition it is possible to modify the structure dynamically during evaluation, which makes various optimizations possible.

King and Moody[12] describe the use of coroutines in CODD in more detail. It is sufficient to note here that the coroutine structure is initialized using Callco and Cowait to inform each node of the identity of its neighbours, but that evaluation uses Resumeco, this being necessary because the evaluation graph will not, in general, be a tree.

As a final example, the coroutine mechanism was used with great success by Stroustrup[14] in an operation system simulator in which he used coroutines to model processes in the operating system. He found the mechanism convenient to use and adequately efficient for his needs.

ACKNOWLEDGEMENTS

The original idea for this coroutine mechanism arose in discussion with Mike Gerard at CERN in Switzerland in 1976. A year later it was reworked at Cambridge after discussions with Bjorne Stroustrup and Tim King who also pointed out the need for Resumeco.

APPENDIX

Now follows an implementation of the coroutine procedures which serves as a guide to people desiring them for their own BCPL systems. The implementation given here is somewhat machine-dependent since it makes assumptions about the allocation of local

variables and the size of the link area in a procedure stack frame. It relies on a special function Changeco. The call Changeco(A, Cptr) suspends the current coroutine (Currco) by saving its resumption point in !Currco; it then sets Currco to Cptr (the new coroutine) and extracts the new resumption point before finally returning with result A to the point just after the call. Thus, on return, execution continues in a new environment which belongs, in general, to a different procedure body in a different coroutine activation. The call is therefore only valid in carefully controlled contexts such as in a RESULTIS command at the outermost level of a function. Its use in Createco is subtle and can only be fully understood by those who have detailed knowledge of the implementation of BCPL.

```
LET Createco(F, Size) = VALOF
$( LET C = Getvec(Size+6)
   IF C=0 RESULTIS 0

   // Using P to denote the current stack frame
   // pointer, the following assumptions are made:
   //    P!0, P!1 and P!2 contain return link information
   //    P!3              is the variable F
   //    P!4              is the variable Size
   //    P!5              is the variable C

   // Now make the vector C into a valid BCPL
   // stack frame containing copies of F, SIZE
   // and C in the same relative positions.
   // Other locations in this new stack frame
   // are used for system purposes.
   C!0 := C        // resumption point
   C!1 := Currco   // parent link
   C!2 := Colist   // Colist chain
   C!3 := F        // main procedure
   C!4 := Size     // coroutine size
   C!5 := C        // the new coroutine pointer

   Colist := C  // insert into the list of coroutines

   Changeco(0, C)
   // execution now continues using C as the stack frame
   // pointer.  The compiler will have generated code on
   // the assumption that F and C are the third and fifth
   // words of the stack frame, and, since C!3 and C!5
   // were initialized to F and C, the following repeated
   // statement will have the effect (naively) expected.
   // Note that the first call of Cowait causes a return
   // from Createco with result C.

   C := F(Cowait(C)) REPEAT
$)

AND Deleteco(Cptr) = VALOF
$( LET A = @Colist
   UNTIL !A=0 | !A=Cptr DO A := !A+2
   IF !A=0 RESULTIS FALSE  // coroutine not found
   UNLESS Cptr!1=0 DO Abort(112)
   !A := Cptr!2   // remove the coroutine from Colist
   Freevec(Cptr)  // free the coroutine stack
   RESULTIS TRUE
$)

AND Callco(Cptr, A) = VALOF
$( UNLESS Cptr!1=0 DO Abort(110)
   Cptr!1 := Currco
   RESULTIS Changeco(A, Cptr)
$)
```

```
AND Cowait(A) = VALOF
$( LET Parent = Currco!1
   Currco!1 := 0
   RESULTIS Changeco(A, Parent)
$)

AND Resumeco(Cptr, A) = VALOF
$( LET Parent = Currco!1
   Currco!1 := 0
   UNLESS Cptr!1=0 DO Abort(111)
   Cptr!1 := Parent
   RESULTIS Changeco(A, Cptr)
$)
```

REFERENCES

1. M. Richards, 'BCPL—A tool for compiler writing and systems programming', *Proceedings of the Spring Joint Computer Conference 1969,* 557–566 (1969).
2. M.Richards and C. Whitby-Strevens, *BCPL–The Language and its Compiler,* Cambridge University Press, 1979.
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1978.
4. R. S. C. Braga, 'Eh reference manual', *Research Report CS–76–45,* Department of Computer Science, University of Waterloo, Ontario, Canada, 1976.
5. W. A. Wulf, D. B. Russell and A. N. Habermann, 'BLISS: A language for systems programming', *Comm. ACM,* **1**, 12 (1971).
6. R. Grune, 'A view of coroutines'. *ACM SIGPLAN Notices,* **12**, 7 (1977).
7. T. Jacobsen, 'Another view of coroutines', *ACM SIGPLAN Notices,* **13**, 4 (1978).
8. W. M. Gentleman, *A Portable Coroutine System,* Information Processing 1971, North Holland Publishing Company, Amsterdam, 1972, p.419–424.
9. B. Krieg. *A Class of Recursive Coroutines,* Information Processing 1974, North Holland Publishing Company, Amsterdam, 1974, p.408–412.
10. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System,* Elsevier Scientific Publishing Company, 1979
11. M. Richards, A. R. Aylward, P. B. Bond. R. D. Evans and B. J. Knight, 'TRIPOS—A portable operating system for mini-computers', *Software—Practice and Experience,* **9**, 513–526 (1979).
12. T. J. King and J. K. M. Moody, 'The design and implementation of CODD'. To be published (1980).
13. S. P. J. Todd, 'The Peterlee relational test vehicle—a sytem overview', *IBM Systems Journal,* **12**, 4 (1976).
14. B. J. Stroustrup, *Communication and Control in Distributed Computer Systems,* Ph.D. Thesis, Cambridge University, 1979.