A low-cost implementation of coroutines for C

Paul A. Bailes

University of Wollongong

A LOW-COST IMPLEMENTATION OF COROUTINES FOR C

Paul A. Bailes

Department of Computing Science
University of Wollongong

Preprint No. 83-9 **November 16, 1983**

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-282-981
telex AA29022

A Low-Cost Implementation of Coroutines for C

Paul A. Bailes

Department of Computing Science

University of Wollongong

Wollongong N.S.W. 2500

Australia

ABSTRACT

We identify a set of primitive operations supporting coroutines, and demonstrate their usefulness. We then address their implementation in C according to a set of criteria aimed at maintaining simplicity, and achieve a satisfactory compromise between it and effectiveness. Our package for the PDP-11 under UNIX† allows users of coroutines in C programs to gain access to the primitives via an included definitions file and an object library; no penalty is imposed upon non-coroutine users.

October 6, 1983

†UNIX is a Trademark of Bell Laboratories.

A Low-Cost Implementation of Coroutines for C

Paul A. Bailes

Department of Computing Science

University of Wollongong

Wollongong N.S.W. 2500

Australia

SUMMARY

We identify a set of primitive operations supporting coroutines, and demonstrate their usefulness. We then address their implementation in C according to a set of criteria aimed at maintaining simplicity, and achieve a satisfactory compromise between it and effectiveness. Our package for the PDP-11 under UNIX† allows users of coroutines in C programs to gain access to the primitives via an included definitions file and an object library; no penalty is imposed upon non-coroutine users.

KEY WORDS C, Coroutines, Language Extension

INTRODUCTION

The purpose of this document is to present an implementation of coroutines for the C programming language [1]. The overriding goal has been to do so with minimal effort and complexity. Advantages of such an approach generally include

(a) the very existence of a result

(b) greater confidence in its correctness.

A further consideration has been transparency - that non-users of coroutines should be as little as possible affected in their use of C by the existence of coroutine facilities. Thus the price of any overheads of, or effects of errors in, the coroutine system will be avoided.

†UNIX is a Trademark of Bell Laboratories.

Pascal [2] is used extensively to describe coroutine-related concepts in abstract terms, because of its status as a *lingua franca* in contemporary computing, and because C syntax tends to be rather too spartan to be comfortably adopted for use in a tutorial context.

COROUTINES

A coroutine [3] is a process, the execution of which, in the monoprogramming environments that are of interest to us, may be suspended in order to initiate or resume the execution of another process, and which itself can be resumed upon the suspension of another.

Abstracting from the facilities found in Simula67 [4], we identify the following requirements.

(a) The ability to retain references to processes, by variables of an appropriate type e.g.

```
var
  a, b: process;
```

declares variables *a* and *b* to contain values which are process references.

(b) Templates or (sub-) programs out of which processes, as the dynamic executions of static program text, may be instantiated. Because we wish to be able to vary the processes instantiated from a single template, a template should accommodate the provision of parameters at instantiation time. For example:

```
template x (fp1, ..., fpn);

  B
```

defines *x* as a template with formal parameters *fp1*, ..., *fpn* and body *B*, in which the parameter names are bound.

(c) A mechanism to perform the instantiation of a process from a template and yield a reference to it:

```
new x (ap1, ..., apn)
```

instantiates a process from template *x* with actual parameters *ap1*, ..., *apn*;

```
b := new x (ap1, ..., apn)
```

serves to retain the reference to the new process in the variable *b*.

(d) A facility to suspend the current process, and take up another:

```
resume b
```

takes up execution of the process referred to by variable *b*.

Instantiation may require initialising variables and data structures local to a process, and so involves the suspension of the instantiating process and the commencement of execution of the body of the template. When the initialisation is complete, the instantiating process is resumed, apparently as a return from the application of the **new** primitive. Therefore, we include

```
    detach
```

as a primitive, which is equivalent to

```
    resume p
```

where *p* is the instantiating or "parent" process.

USE OF COROUTINES - EXAMPLE

Inspired by Knuth [5], coroutines may be seen to advantage in the following situation. An input to a process P consists of atoms structured in some way (e.g. grouped in pairs), and the output is required to consist of the atoms structured in some other way (e.g. grouped in threes). A simple solution may be arrived at by decomposing P into P1 and P2: P1 decomposes a sequence of pairs into a sequence of atoms, and P2 accepts this output from P1 and assembles threes.

Assuming for the sake of simplicity that the atoms are integer numbers, and that sequences of atoms or groups thereof are represented by files, then the definition of P in Pascal (assuming a predefined type *string*) would appear as in Program 1:

```
program P (ifile, ofile, tmpfile);

var
   tmpfile : file of integer;

procedure P1 (iname : string);

   type
      pair = array [0..1] of integer;

   var
      ibuf : pair;
      ifile : file of pair;

   begin
   reset (ifile, iname);
   rewrite (tmpfile);
   while not eof (ifile) do
      begin
      read (ifile, ibuf);
      write (tmpfile, ibuf [0]);
      write (tmpfile, ibuf [1])
      end
   end;

procedure P2 (oname : string);

   type
      three = array [0..2] of integer;

   var
      obuf : three;
      ofile : file of three;
      ocount : integer;

   begin
   reset (tmpfile);
   rewrite (ofile, oname);
   ocount := 0;
   while not eof (tmpfile) do
      begin
      read (tmpfile, obuf [ocount]);
      ocount := (ocount + 1) mod 3;
      if ocount = 0 then
         write (ofile, obuf)
      end
   end;

begin
P1 ('input.data');
P2 ('output.data')
end.
```

Program 1

If the number of atoms is not divisible by three, then either the last or the last two will be missed. Let us accept this.

Note, however, the introduction of a third, intermediate file called *tmpfile*. This inefficient use of storage would be removed if *P1* were to deliver only its first three results (as members of the intermediate file) before *P2* were to process and output them, after which *P1* would deliver the next three, and so on. Evaluation of function invocation and list processing operations (for files can be thought of as lists on secondary storage) under a delayed or lazy [6, 7] regime would achieve this automatically, but is only viable in an applicative context. Coroutines, where the lock-step relationship of producer and consumer is indicated explicitly, provide a solution.

Augmenting Pascal with our suggested primitives would allow, for example, Program 2:

```
program P (ifile, ofile);

var
   endfile : boolean;
   tmp : integer;
   P1, P2 : process;

template inproc (var outp : process; iname : string);

   type
      pair = array [0..1] of integer;

   var
      ibuf : pair;
      ifile : file of pair;

   begin
   endfile := false;
   reset (ifile, iname);
   detach;
   while not eof (ifile) do
      begin
      read (ifile, ibuf);
      tmp := ibuf [0];
      resume outp;
      tmp := ibuf [1];
      resume outp
      end;
   endfile := true;
   resume outp
   end;

template outproc (var inp : process; oname : string);

   type
      three = array [0..2] of integer;

   var
      obuf : three;
      ofile : file of three;
      ocount : integer;

   begin
   rewrite (ofile, oname);
   ocount := 0;
   detach;
   while not endfile do
      begin
      obuf [ocount] := tmp;
      ocount := (ocount + 1) mod 3;
      if ocount = 0 then
         write (ofile, obuf);
      resume inp
      end;
   detach
   end;
```

```
begin
P1 := new inproc (P2, 'input.data');
P2 := new outproc (P1, 'output.data');
resume P1
end.
```

Program 2

Each process is instantiated with a reference to the variable referring to the other as a parameter. Thus, when for example, we refer to *outp* inside *inproc*, this is equivalent to a reference to *P2*, the instantiation of *outproc*.

By way of comparison, we present Program 3 as a solution to the problem, avoiding the use of temporary files and eschewing coroutines:

```
program P (ifile, ofile);

type
   pair = array [0..1] of integer;
   three = array [0..2] of integer;

var
   endfile : boolean;
   tmp, icount, ocount : integer;
   ibuf : pair;
   ifile : file of pair;
   obuf : three;
   ofile : file of three;

procedure P1;
   begin
   case icount of
      0:
         if eof (ifile) then
            endfile := true
         else
            begin
            icount := 1;
            read (ifile, ibuf);
            tmp := ibuf [0]
            end;
      1:
         begin
         icount := 0;
         tmp := ibuf [1]
         end
      end
   end;

procedure P2;
   begin
   obuf [ocount] := tmp;
   ocount := (ocount + 1) mod 3;
   if ocount = 0 then
      write (ofile, obuf)
   end;

begin
endfile := false;
reset (ifile, 'input.data');
icount := 0;
rewrite (ofile, 'output.data');
ocount := 0;
P1;
while not endfile do
   begin
   P2;
   P1
   end
end.
```

Program 3

In Program 3 modularity is degraded. In Program 2, all the detail of reading pairs from a named file is captured in a single coroutine template, and likewise for writing threes to a named file, with only the communications between the two being visible externally (variables *tmp* and *endfile*). Without coroutines it is necessary to make visible the variables logically local to the (abstract) processes P1 and P2, but which need to be preserved over a series of procedure invocations. Furthermore, initialisation involving parameters (in this case, the name of the file) which one does not wish to specify for each procedure invocation, needs to be separated from the remainder of the code for the process. It can be expressed either in-line, as above, or at best as a separate procedure whose call needs to be made explicitly.

While Program 2 using coroutines may be longer, it is factored into manageable sub-programs. Of course, there have been proposed facilities (e.g. [8]) which seek to directly and exclusively address scope issues as raised above, and allow the hiding of names whose associated variables may persist over several calls to a procedure or function. Even so, we still require the introduction of "state variables" such as *icount* above. We submit that coroutines provide a mechanism to address the issues of both scope and the introduction of new variables.

LANGUAGE EXTENSION

Having now established the desirability of coroutines, we address the issue of making them available in languages where the relevant facilities are not provided. The "obvious" approach, of producing a new compiler (from scratch, or by modifying an existing one), is at odds with the policy adopted in the **INTRODUCTION**. We look to the field of language extension for relief. Standish [9] provides the following taxonomy of extension mechanisms.

(a) Paraphrasic extension means a new entity is given a meaning in terms of the composition of entities already defined or available. Procedures and functions are examples of paraphrasic extension mechanisms found in many programming languages.

(b) Orthophrasic extension means a new entity is defined by means other than those already provided. Since most programming languages can express all the computable functions, then by Turing's thesis etc., no extension in such a language can be *inherently* orthophrasic - it is the *way* in which the extension is made that is of interest.

(c) Metaphrasic extension is related to orthophrase in that an existing notation or entity is altered e.g. redefining the arithmetic operators to accommodate complex arithmetic.

Given our insistence on a low-cost implementation, the provision of coroutines via exclusively paraphrasic extensions must be ruled out for the following reasons. The essence of implementing coroutines is how the concept of *process* is to be represented. In abstract terms, this is by a pair

(data, state)

where *data* is a reference to the storage local and unique to the process, and *state* indicates the next piece of code to be executed. Procedures (or functions, in C terminology) alone will not suffice - though we may have more than one activation of the same procedure at any one time (via recursion), the last-in first-out regime imposed on storage allocation is too restrictive for coroutines.

Consequently, space for local data would have to be explicitly allocated as part of process instantiation, and references to local variables (including value parameters) changed to offsets of references to the start of this space. Given that *D* denotes this, the *data* component of some process, then, for example, the C statement

```
x = 3;
```

where *x* is a local variable, would become in a paraphrasic implementation of coroutines for C:

```
D [size of storage for locals allocated prior to x] = 3;
```

As for the *state* component, this is automatically accounted for by the program counter while a process is executing. However to suspend a process would involve the setting of its *state* to indicate where to begin execution after re-activation. This may be facilitated by writing the body of the process template as a multi-way branch, one for each possible point of re-activation of the process, and branching on the value of a *state* indicator to one of these upon re-activation. This was done implicitly in writing procedure *P1* in Program 3 above.

While this analysis is by no means a complete specification of the design of a coroutine system by paraphrasic extension, it does indicate that a considerable amount of preprocessing would be needed to provide an acceptable syntactic interface to the programmer using coroutines. As indicated earlier, we are not prepared to undertake a compiler writing exercise. For an ortho-metaphrasic approach, the task of modifying the behaviour of procedures to support processes is strongly suggested, because procedures so very nearly achieve the level of facility required by coroutines (i.e. allocation of local storage, multiple activation from the one code template). The remainder of the paper discusses such an exercise applied to C.

IMPLEMENTATION - DATA STRUCTURES

For each process there will be needed the (*data*, *state*) pair. We define

```
struct __cstruct
{
unsigned __cdata, __cstate;
struct __cstruct *__cparent;
};
```

where *__cdata* and *__cstate* will, in a manner to be described, denote the *data* and *state* components required. The *__cparent* field is a reference to the parent of the process in question, in order to facilitate the implementation of the **detach** primitive.

When a program executes, the initial process associated with the "main" program exists by default, and a structure of the kind just described must be made available for it.

Defining

```
typedef struct __cstruct *process;
```

allows us to describe references to process data structures via the name *process*. We subsequently have

```
struct __cstruct __cmain;

process __cproc = &__cmain;
```

which defines *__cmain* as the process structure for the initial process, and which defines *__cproc*, the role of which is that of a global variable identifying the current process, to initially refer to the initial process.

IMPLEMENTATION - PRIMITIVE OPERATIONS

The coroutine primitive operations (identified in our system by the names *NEW*, *RESUME* and *DETACH*) are all initially implemented as macros. The abstract

proc := **new** template (args)

is effected by the call

```
NEW (proc, template (args))
```

where

```
        #define NEW(proc, call) if (__savproc (__cproc))\
        {proc = __cgetsp (); (proc)->__cparent = __cproc; __cproc = proc; call;}
```

The function *__savproc* preserves (*data*, *state*) information about the current process in the space referred to by its argument (*__cproc*). Accepting for the moment that this call of *__savproc* will yield a true (non-zero) result, space is allocated (via the call on *__cgetsp*) for the new process, the pointer to which is stored in the provided variable *proc* identifying the new process. The parent of the new process is the currently executing process, the identity of which is assigned as the *__cparent* of the new process structure. Then, the new process is identified as the current process, and the call on the template made to allocate local storage and begin execution.

The abstract

resume proc

is implemented by the call

```
        RESUME (proc)
```

where

```
        #define RESUME(proc) if (__savproc (__cproc))\
        {__cproc = proc; __rstproc (__cproc);}
```

After preserving information about the (old) current process, the (new) current process is identified as that to be **resumed**, and it is reactivated by the call on *__rstproc*. Reactivation of a suspended process appears as a false (i.e. zero) return from the call of *__savproc* which suspended it.

Finally,

detach

is simply effected by the call

```
        DETACH
```

where

```
        #define DETACH if (__savenv (__cproc))\
        {__cproc = (__cproc)->__cparent; __rstproc (__cproc);}
```

i.e. *DETACH* is the same as *RESUME* except that it is the parent of the current process that is re-activated.

IMPLEMENTATION - AUXILIARY PROCEDURES

Function *__cgetsp* allocates and returns a pointer to space for a process activation record:

```
process __cgetsp ()
   {
   process tmp;

   if ((tmp = malloc (sizeof (*tmp))) == NULL)
      {
      fprintf (stderr, "no room for process activation record\n");
      exit (1);
      }
   else
      return tmp;
   }
```

It uses the standard function *malloc* to allocate space, and aborts with an appropriate diagnostic if the space cannot be found.

Before discussing *__savproc* and *__rstproc*, it is necessary to discuss just what needs to be saved and restored. Note first that the coroutine primitives are implemented as statement-level rather than expression-level constructs, and that the C compiler contemplated in this exercise (that for the PDP-11) does not maintain storage for temporaries on the stack or in registers between statements. Consequently the *data* component can be catered for by preserving the current stack frame pointer, referring to the local storage of the invocation of the function which is the template for the process in question. As for *state*, all that is required is to save the address of the instruction to be executed upon re-activation.

Thus, for the PDP-11, we define *__savenv*:

```
mov   r5,       *2(sp)  ;save frame pointer (r5)
                        ;in first word of space
                        ;pointed to by argument
                        ;i.e. the data component
                        ;of the process space

add   $2,       2(sp)   ;increment argument by 2
                        ;to point to the next word
                        ;in the process space

mov   (sp),     *2(sp)  ;save the contents of the
                        ;current top of stack, i.e.
                        ;the return address for this
                        ;call of __savenv, in the
                        ;state component

mov   $1,       r0      ;make the function result
                        ;(r0 contents) true

rts   pc                ;return from call
```

We correspondingly define *__rstenv*:

```
mov   *2(sp),   r5      ;restore frame pointer

add   $2,       2(sp)   ;increment argument

mov   *2(sp),   r1      ;place in r1 return address
                        ;of the call of __savenv that
                        ;suspended the process we are
                        ;reactivating

;now simulate the false return from __savenv

clr   r0                ;function result false

add   $2,       sp      ;pop stack

jmp   r1                ;jump to required return address
```

DISCUSSION

The essence of the implementation is that process instantiation is effected by the simple call of the template function, with space for local storage being allocated as usual on the stack. The *data* component of the process is given by the frame pointer for this stack frame. It is saved when a process is suspended. When a suspended process is re-activated, the frame pointer is re-set according to the *data* component of its activation record, but the stack pointer is unaffected. For example, if the initial (main) process instantiates processes A and B from templates X and Y, and upon re-activation invokes procedure Z, the stack appears as in Fig. 1:

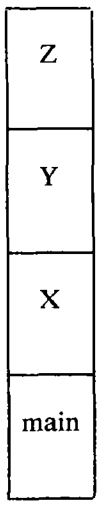

Figure 1

Maintaining the integrity of the scheme is the cause of restrictions, both upon implementations of C with which it is compatible and upon programs written within it. Fundamentally, a process which pushes information onto the stack must be that which pops the same elements. With respect to implementations of C, if, for example, process P1 pushes a temporary T1, and then process P2 pushes a temporary T2, and then P1 is re-activated, P1 may not now attempt to dispense with the space for T1. In other words, the compiler may not maintain over a coroutine primitive operation stack storage for temporaries. Because this use of stack storage is typically the consequence of exceeding available register storage for temporaries, we therefore exclude all consideration of maintaining temporary storage under such circumstances. Coroutine primitives are statement-level constructs, and thus the PDP-11 compiler is compatible because it does not maintain temporaries between statements.

Similar restrictions apply to the ways in which functions may be called and returned from. Let us modify Fig. 1 above, to a stage where process A has called function F1, which then suspends to resume process B, which in turn calls function F2. Fig. 2 represents the current situation.

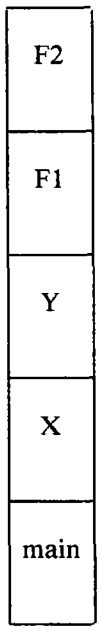

Figure 2

Process A is represented by the space for X and Fl, process B by that for Y and F2. If process B, during the activation of F2, suspends to resume process A, then function Fl may not return, as this does not operate the stack in the required last-in first-out manner.

A related restriction is that functions invoked as coroutine instantiations may never be returned from. This is because it would lead to the instantiating process being resumed from the point just after which it *instantiated* the just-terminated process. In the light of the above warnings, this does not seem to be too additionally confining. It follows that any functions active when a new process is instantiated may also never be returned from.

Violations of these restrictions may only be avoided by programmer discipline. One which is more restrictive than necessary, but which is easy to remember, is to only use coroutine primitive operations in the *main* function of the C program or in functions directly invoked as coroutine instantiations.

COMPARISONS

Gentleman[10] has proposed a scheme for FORTRAN[11] which meets our criteria. Coroutine primitives are effected by calls on subroutines, some of which are written in machine code. Significantly, for a language whose storage allocation is statically-oriented, the coroutines are true process templates. Processes instantiated from the same template may be given unique data space.

However, this allocation has to be done explicitly by the programmer, which makes the interface a little clumsy.

Another scheme [12] applies paraphrase to FORTRAN. A preprocessor allows a suitable syntactic interface, producing a program with state variables and multi-way branches as suggested above. Coroutines are not templates for processes - only one process may be associated with a piece of code, and thus the problems (and benefits) of providing unique data space are avoided.

An implementation [13] of coroutines for BCPL [14] corresponds with our approach. Several important differences are as follows.

(a) In this system, unlike ours, the relationship between processes is strictly hierarchical. A process which suspends to invoke another is distinguished with respect to the invoked process. Under our scheme, the only hierarchy recognised is that of one process with respect to those which it instantiates.

(b) The BCPL scheme allows for the reclamation of storage allocated to a process when its execution terminates. In contrast, we do not permit a function invoked as a process instance to be returned from.

(c) The BCPL scheme requires that the size of storage local to an instantiated process be passed as an argument to the procedure performing the instantiation. This means that the programmer must calculate and write down such a number in his/her program. In contrast, storage allocation in our scheme is automatic.

Finally, there exists [15] a coroutine scheme for Pascal which shares some properties of ours and/or that just described. Processes may be instantiated from templates, as in both those schemes, but are strictly non-hierarchical - even the instantiating parent of a process is not distinguished (unlike ours). On the other hand, local process storage has to be explicitly allocated, with the programmer providing the size of the required space. A facility to reclaim space on termination is provided.

In summary, our scheme is distinct from all of the above. As well as implementing coroutines for a different language, it embodies a distinct combination of general support for the abstract model of coroutines introduced at the beginning of this document, together with an unobtrusive set of interface procedures.

UNIX INTERFACE

Three files are required for the system. The first, *corout.h* is a file of definitions that should be included at the head of each C source file using the coroutine primitives:

```
struct __cstruct
{
unsigned __cdata, __cstate;
struct __cstruct *__cparent;
};

typedef struct __cstruct *process;

#define NEW(proc, call) if (__savproc (__cproc))\
{proc = __cgetsp (); (proc)->__cparent = __cproc; __cproc = proc; call;}

#define RESUME(proc) if (__savproc (__cproc))\
{__cproc = proc; __rstproc (__cproc);}

#define DETACH if (__savenv (__cproc))\
{__cproc = (__cproc)->__cparent; __rstproc (__cproc);}
```

The second, *cgetsp.c*, contains the definition of the *__cgetsp* function and initialises *__cproc*:

```
#include <stdio.h>

struct __cstruct
{
unsigned __cdata, __cstate;
struct __cstruct *__cparent;
};

typedef struct __cstruct *process;

struct __cstruct __cmain;

process __cproc = &__cmain;

process __cgetsp ()
   {
   process tmp;

   if ((tmp = malloc (sizeof (*tmp))) = = NULL)
      {
      fprintf (stderr, "no room for process activation record\n");
      exit (1);
      }
   else
      return tmp;
   }
```

The third, *savrst.s*, contains the definitions of *__savproc* and *__rstproc*:

```
____savenv:
        mov     r5,     *2(sp)
        add     $2,     2(sp)
        mov     (sp),   *2(sp)
        mov     $1,     r0
        rts     pc

____rstenv:
        mov     *2(sp), r5
        add     $2,     2(sp)
        mov     *2(sp), r1
        clr     r0
        add     $2,     sp
        jmp     r1
```

The object files of the latter two should be link-edited with any C program using coroutines. The only names that user programs should access are *process*, *NEW*, *RESUME* and *DETACH*. The remaining visible names all have the prefix *__c* to reduce the risk of collision with a user-defined name.

If *corout.h* is installed in the directory */usr/include*, then the line

```
#include <corout.h>
```

will include the definitions file. If the object files of *cgetsp.c* and *savrst.s* are combined in the object library file */lib/libC.a*, then for example to compile the coroutine-using C program *test.c* to give the executable program *test*, the command

```
cc -o test test.c -lC
```

will suffice.

EXAMPLE

Program 2 above would appear in our C system as follows:

```
#include <stdio.h>
#include <corout.h>

int endfile, tmp;

process P1, P2;

inproc (outp, iname)        /* template */
process *outp;
char *iname;
   {
   int ibuf [2];
   FILE *ifile;

   endfile = 0;
   ifile = fopen (iname, "r");
   DETACH;
   while (! eof (ifile))
      {
      getpair (ifile, ibuf);
      tmp = ibuf [0];
      RESUME (*outp);
      tmp = ibuf [1];
      RESUME (*outp);
      }
   endfile = 1;
   RESUME (*outp);
   }

outproc (inp, iname)        /* template */
process *inp;
char *oname;
   {
   int obuf [3], ocount;
   FILE *ofile;

   ofile = fopen (oname, "w");
   ocount = 0;
   DETACH;
   while (! endfile)
      {
      obuf [ocount] = tmp;
      ocount = (ocount + 1) % 3;
      if (ocount = = 0)
         putthree (ofile, obuf);
      RESUME (*inp);
      }
   DETACH;
   }

main ()
   {
   NEW (P1, inproc (&P2, "input.data"));
   NEW (P2, outproc (&P1, "output.data"));
   RESUME (P1);
   }
```

Points to note are

(a) the assumption that it is possible to augment the facilities of the standard I-O package to accommodate three extra functions: *getpair* which reads a pair of integers from a file and places them in an array; *putthree* which outputs three integers from an array to a file; and *eof*, which tests to see if the end of a file containing pairs of integers has been reached

(b) the simulation of reference parameters for the *process* parameters *P1* and *P2* by the usual C device of passing addresses as actual parameters and referring to pointers as formal parameters.

CONCLUSIONS

The system presented above achieves a workable implementation of the coroutine concept. It achieves its design goals, of simplicity and of avoidance of interference with non-coroutine users and compares favourably with similar enterprises.

ACKNOWLEDGEMENTS

I am grateful to the following for their contributions to this work: Eric Salzman, for providing the initial stimulus; Robert Enchelmaier, for his description of the PDP-11 and its C compiler; and Ross Nealon, for a similar task with respect to the Perkin-Elmer 3200 series. My thanks also go to Lynn Maxwell for typing this manuscript.

REFERENCES

1. Kernighan, B.W. and Ritchie, D.M., "The C Programming Language", Prentice-Hall, 1978
2. Jensen, K. and Wirth, N., "Pascal User Manual and Report", Springer, 1974
3. Conway, M.E., "The Design of a Separable Transition-Diagram Compiler", CACM, Vol 6, pp 396-408, 1963
4. Dahl, O-J., Myhrhang, B. and Nygaard, K., "The SIMULA 67 Common Base Language", Publication S-22, Norwegian Computing Centre, Oslo, 1968
5. Knuth, D.E., "The Art of Computer Programming Volume 1 Fundamental Algorithms", Addison-Wesley, 1968

6. Friedman, D.P. and Wise, D.S., "CONS Should Not Evaluate Its Arguments", in "Automata, Languages and Programming", S. Michaelson and R. Milner (eds.), Edinburgh University Press, 1976

7. Henderson, P. and Morris, J.H., "A lazy evaluator", Proceedings 3rd ACM Symposium on Principles of Programming Languages, pp 95-103, 1976

8. Wirth, N., "The Module: A System Structuring Facility in High-Level Programming Languages", Proceedings Symposium on Language Design and Programming Methodology, Sydney, 1979

9. Standish, T.A., "Extensibility in Programming Language Design", Proceedings National Computing Conference pp 386-390, 1975

10. Gentleman, W.M., "A Portable Coroutine System", Information Processing 71, pp 419-424, 1971

11. ANSI Standard Fortran, Publication X.39, American National Standards Institute, 1966

12. Skordalakis, E. and Papakonstantinou, G., "Coroutines in FORTRAN", SIGPLAN Notices, Vol. 13, No. 9, pp 76-84, 1978.

13. Moody, K. and Richards, M., "A Coroutine Mechanism for BCPL", SOFTWARE - Practice and Experience, Vol 10, pp 765-771, 1980

14. Richards, M., "BCPL - A tool for compiler writing and systems programming", Proc. Spring Joint Comp. Conf. 1969, pp 557-566, 1969

15. Kriz, J. and Sandmayr, H., "Extension of Pascal by Coroutines and its Application to Quasi-Parallel Programming and Simulation", SOFTWARE - Practice and Experience, Vol 10, pp 773-789, 1980