stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 63/76 NOVEMBER

D. GRUNE

A VIEW OF COROUTINES

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS(MOS) subject classification scheme (1970): 68A05

ACM-Computing Reviews-categories: 4.10, 4.20, 4.32

A View of Coroutines[*)]

by

D. Grune

ABSTRACT

The coroutine mechanism is explained as a simplified implementation of a special case in parallel processing.

KEYWORDS & PHRASES: Coroutines, parallel processing, SIMULA 67, ALGOL 68.

*) This report will be submitted for publication elsewhere.

1. Introduction.

Experience has shown that the coroutine mechanism is an order of magnitude harder to understand and explain than the subroutine mechanism, and I have always wondered why. The following thoughts, though incomplete, may shed some light on the subject.

There is one immediately striking result of the discrepancy in difficulty: hardly any modern programming language provides facilities for coroutine calling, although some make an attempt [1]. So it is only natural that this paper arose in a study of the flow of control in existing programming languages [4].

Both subroutines and coroutines rely for their flow of control on return addresses; these return addresses are addresses in the calling routine, to where the called routine must (ultimately) return. In the case of a subroutine the return address is kept with the called routine or, in the recursive case, on a stack or stack segment that belongs to the called subroutine.

In the case of a coroutine, however, the return address is kept with the calling routine, or, in the recursive case, on a stack that belongs to the calling routine. Recursive coroutines are exceedingly rare [2, 3]; it is not clear whether this is so because of conceptual difficulties or for lack of practical use.

The above, though true, adds little to the understanding of coroutines. An example is therefore in order. As Knuth remarks [3], it is rather difficult to find short, simple, illustrative examples of applications of coroutines. Since I shall take the program apart and put it together again in several languages I shall need a very simple application indeed. The following highly contrived example will do.

We have a process A which copies characters from input to output with the proviso that where the input has "aa" the output will have "b" instead. And we have a similar process B which converts "bb" into "c". Now we want to connect these processes in series by feeding the output of A into B. In order do to this, we could consider B as the main program and have it call the subroutine A for each character. But the process A, in its most reasonable form, contains 4 calls of the output routine in various places. If we turn A into a subroutine, all 4 of them have to be moved to the end (where the character is to be delivered) which means major surgery to A. So we prefer to keep A as a main program along with B, by connecting them through a coroutine link. (A as a main program with B as subroutine is, of course, no better).

2. The low-level version.

The following machine-code program, coded in ALGOL 68 [5], shows the mechanism.

```
begin proc void label A:= proc A, label B:= skip;
  proc void
    co call A = (proc void L) void:(label B:= L; label A),
    co call B = (proc void L) void:(label A:= L; label B);
  char ch, ch1;

goto proc B;

  proc A:
    read(ch); if ch = "a" then goto a found fi;
    co call B(L2); L2: goto proc A;
  a found:
    read(ch); if ch = "a" then goto generate b fi;
    ch1:= ch; ch:= "a"; co call B(L3); L3:
    ch:= ch1; co call B(L4); L4: goto proc A;
  generate b:
    ch:= "b"; co call B(L5); L5: goto proc A;

  proc B:
    co call A(L6); L6: if ch = "b" then goto b found fi;
    print (ch); goto proc B;
  b found:
    co call A(L7); L7:if ch = "b" then goto generate c fi;
    print ("b"); print (ch); goto proc B;
  generate c:
    print ("c"); goto proc B

end
```

Explanation:
The void-procedures "label A" and "label B" record the position in "proc A" and "proc B" respectively: "co call A" (executed from "proc B") obtains the position in "proc B" as its parameter, stores it in "label B" and causes "proc A" to continue in the position "label A" (and vice versa).

This program invites several objections.

- I. We had to carve up processes A and B for their internal positions to become available to the coroutine mechanism in the form of labels.

- II. We had to introduce the labels L2 through L7 which contribute less than nothing to clarity and ease of programming.
- III. The text makes in no way evident the cardinal point for each coroutine request: the information transfer through variable "ch". Each time "proc A" has a new character ready it does a "co call B", and conversely, each time "proc B" needs a character, it does a "co call A".
- IV. Nothing in the above notation prevents us from doing a "co call A" from inside "proc A", and thus wreck the flow of control.
- V. The initializations of "label A" and "label B" and the initial jump to "proc B" are opaque.

The above shows that it is possible to write coroutines in ALGOL 68 and at the same time makes it clear that this particular method is not a reasonable one.

It is interesting to note in passing what would have happened if I had written the program in traditional coroutine fashion. Then "label A" and "label B" would coincide (since only one is meaningfull at a given instant), coroutine calls would be implemented as address exchanges (an optimization that is valid for two coroutines only, and is extremely confusing in the case of three) and labeled jumps would be shunted out. The resulting program would open up new horizons in unreadability.

3. The Simula 67 version.

The coroutine mechanism is available explicitly in only one major programming language, Simula 67 [1], and it is interesting to see how well it does. Here the program would be:

```
begin character ch;

  class double a to b;
  begin detach;
    while true do
    begin ch:= inchar;
      if ch = "a" then
      begin ch:= inchar;
        if ch = "a" then
        begin ch:= "b"; resume (proc B) end
        else begin character ch1; ch1:= ch;
             ch:= "a"; resume (proc B);
             ch:= ch1; resume (proc B)
           end
      end else resume (proc B)
    end infinite loop
  end double a to b;
```

```
    class double b to c;
    begin detach;
      while true do
      begin
        if ch = "b" then
        begin resume (proc A);
          if ch = "b" then outchar ("c") else
          begin outchar ("b"); outchar (ch) end
        end else outchar (ch);
        resume (proc A)
      end infinite loop
    end double b to c;

    ref (double a to b) proc A;
    ref (double b to c) proc B;

    proc A :- new double a to b;
    proc B :- new double b to c;
    call (proc A)
  end
```

Explanation:
The construction delineated by class ... begin ... end defines a class of processes each of which performs the actions described between the begin and end. The statements starting with ref declare objects of the indicated classes and each of the two statements containing a :- creates a process of the given class and assigns it to "proc A" or "proc B" respectively. The next statement starts the "proc A". A coroutine call is written "resume (process)". A call of "detach" signals the end of the initialization process, which is performed during creation of the object.

This version is in many ways an improvement over the former; however, most objections still hold, though often in a mitigated form:
- I. The processes more or less retain their original form; although they are conceptually identical, they still have to be quite different textually.
- II. No spurious labels are required.
- III. The transfer of information through the variable "ch" has not yet been given the prominent place it deserves.
- IV. We can still call "resume (proc A)" inside "proc A" (run-time check?).
- V. The initialization is clearer (but traces of the initial jump to "proc B" in the machine-code version are hidden in the flow-of-control of "double b to c").

Moreover, a new objection can be raised. It does not become evident in the given example but emerges in the example given on pages 188-189 of [1]:

- VI. If we connect three processes A, B and C in series, then A calls B in order to _dispose of information_ whereas C calls B in order to _obtain information_, both by the same instruction "resume B". And since the class-declaration gives no indication whatsoever of the use of the process, the semantics of "resume B" is statically obscure.

4. The ALGOL 68 Version.

In view of all these problems it is remarkable that ALGOL 68 allows the desired effect to be obtained by simple application of the otherwise rudimentary features _par_ and _sema_, without the explicit use of coroutines. The program will then be:

```
begin
struct (sema write, ref char ch, sema read) interface =
       (level 1,    loc char    , level Ø);

  proc co write = (char ch) void:
  (down write of interface; ch of interface:= ch;
   up read of interface),

  proc co read = (ref char ch) void:
  (down read of interface; ch:= ch of interface;
   up write of interface);

  par begin  # proc A: #
    do char ch; read (ch);
      if ch = "a"
      then read (ch);
        if ch = "a" then co write ("b")
        else co write ("a"); co write (ch) fi
      else co write (ch)
      fi
    od,
    # proc B: #
    do char ch; co read (ch);
      if ch = "b"
      then co read (ch);
        if ch = "b" then write ("c")
        else write ("b"); write (ch) fi
      else write (ch)
      fi
    od
  end
end
```

The first two lines define an interface consisting of a "ref char ch", enclosed between two barriers, "sema write" and "sema read". In the beginning the barrier "write" is open, "read" is closed.

The next six lines define two routines, "co write" and "co read". The routine "co write" closes the write barrier (or waits if the barrier happens to be down already), copies its parameter "ch" onto the interface and opens the read barrier. "Co read" does the reverse and empties the interface.

Execution of the construction par begin A, B end causes A and B to be executed (pseudo)simultaneously, so that any synchronization will be through semaphores only.

All objections but one have disappeared:

- I. Both processes are in perfect shape now, and identical.
- III. Safeguarded information transfer holds a central place.
- V. The initialization is perfectly clear.
- VI. The information transfer is governed by unambiguous, well-defined procedure-calls.

Two problems remain: we can still call "co read" from within "proc A" (risking deadlock) and we can tinker with the interface. This lack of protection is a sore point in all major programming languages of today.

And in addition to solving most of the problems we have gained something new. From the point of view of program structure the last version expresses our intentions much better; when A calls B for the transfer of a character it does not at all require B to start processing it immediately, A only wants to dispose of it. Only when the interface (pipe-line) gets full must B proceed. This interplay cannot be specified in a coroutine version; but it comes naturally to the semaphore version, which leaves it undefined in exactly the right measure whether A or B shall proceed.

5. Conclusion.

We now perceive the coroutine mechanism as a simplified implementation of a special case in parallel processing. This view opens several new avenues of thought. It facilitates understanding recursive coroutines. Perhaps there are other special cases in parallel programming that allow simplified implementation and result in useful features. Perhaps the above interface mechanism is even so fundamental that it warrants a special construction in parallel processing.

6. References.

[1] Dahl, O-J., et al., Structured Programming, p. 175-220, APIC Studies in Data Processing 8, Academic Press, London, 1972.

[2] Krieg, B., A Class of Recursive Coroutines, Proceedings of IFIP Congress 1974, p. 408-412, North Holland Publ. Company, Amsterdam, 1974.

[3] Knuth, D.E., The Art of Computer Programming, Vol I, p. 190-196, Addison-Wesley Publ. Company, London, 1969.

[4] Grune, D., Flow-of-control, Vergelijking van bestaande programmeertalen, (Flow-of-Control, Comparison of Existing Programming Languages), in MC Syllabus 25, p. 1-20, Mathematical Centre, Amsterdam, 1976 (in Dutch).

[5] van Wijngaarden, A., B.J.Mailloux, J.E.L.Peck, C.H.A.Koster, M.Sintzoff, C.H.Lindsey, L.G.L.T.Meertens & R.G.Fisker (eds), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975) 1-236.