An answer to Jack Mazola.

Thank you for your letter of 7 Nov. 1975. I was very pleased to read that my writings had inspired John McClintock and you to do better than your (or McClintock's) first effort. I had --and I offer my apologies if this is a great disappointment-- difficulties in understanding your invariants, because they are mixed expressions in two different languages: on the left-hand side of the implication we have boolean expression which at any moment in time are defined by the current state of the aggregate, at the right-hand side of the implication are phrases about past and future, and not all sentences --such as "if he ever restarts"-- were too clear to me. The only thing left for me to do was to try to convince myself that everything was OK in my own (old-fashioned! see below) way. The following is not a description of how I would develop the program ab initio: I cannot fake that, I have been "spoiled" by studying your solution and, as a result, have lost my virginity. The first thing I did was to study the following program (here "x" is a local variable of each machine)

```
do non initialized →
        done[me]:= false;
        x:= 1; do done[x] → x:= x + 1 od;
        if x = me → initialized:= true
         [] x < me → skip
        fi;
        done[me]:= true
od
```

and asked myself the question whether "initialized" will become true, when a number of the machines is started. The answer is "Yes", provided the machines that re not started have their done = true; the argument is that of the started machines the one with the lowest number is certain to end its inner loop with x = me and will therefore perform "initialize:= true". Note that for this argument the initial value of done for the machines that are started, is irrelevant.

The next question is: can we make of "initialize:= true" mutually exclusive actions --so-called "critical sections". So I tried to do it by inserting your waiter (note that in your letter, page 4, is a misprint, it should be

"do X ≤ HI → do non DONE[X] → SKIP od;")

```
do non initialized →
      done[me]:= false;
      x:= 1; do done[x] → x:= x + 1 od;
      if x = me → do x < N → x:= x + 1;
                                 do non done[x] → skip od
                  od;
                  initialized:= true
       [] x < me → skip
      fi;
      done[me]:= true
od
```

Assume that at moment t machines i and j (i ≠ j) are both ready to perform "initialized:= true". Then we have at moment t: done[i] = done[j] = false.
Let ti be the last moment < t such that done[i] was true;
let tj be the last moment < t such that done[j] was true.
From the fact that machine i has reached the statement "initialized:= true" we can conclude that done[j] = true has been observed while done[i] was already = false , hence ti < tj . Similarly we find that tj < ti , and this gives us the required contradiction. Mutual exclusion is guaranteed.

The next question is: can we have introduced deadlock by our addition? We cannot have a number of processes mutually blocking each other with their done = false in that added waiting cycle, because the one with the largest number of the set does not inspect the other values done from the set. But machine i may be kept in that inner cycle by a machine j (j > i) happily rotating in the outer cycle, but with such a speed that machine i always misses the rare moments that done[j] = true . Therefore the assignment done[me]:= true is followed by

```
x:= 1;
do x < me → do non done[x] → skip od;
            x:= x + 1
od
```

Because of the arguments given by you this does not introduce the danger of deadlock in the last waiting cycle.

The last modification is to ensure that initialization now occurs only once; because "initialized:= true" has been guaranteed to be executed in a critical section, it suffices obviously to replace it by

do non initialized → initialize; initialized:= true od (1)

and now I have arrived at a program that differs only marginally from your solution, which I thought a very beautiful one.

* * *

There is a second way in which I am no virgin with respect to this problem: it is very similar to the critical section problem that I solved in the Sep.1965 issue of the Comm.ACM., and naturally, the old patterns of reasoning come again floating in my consciousness. To prove things about such processes that may interfere with each other in such a fine-grained fashion is a risky business, as I have learned the hard way.(In the meantime you should have received EWD520, our next, and hopefully last, version of the on-the-fly garbage collection.) The person with the most extensive experience of proving things about such aggregates of programs is David Gries. One of the more valuable tricks is to associate "ghost variables" to which the programs only assign values. They --i.e. their values-- can then be used to express things about the progress of the various programs, they provide means for expressing formally and without ambiguity such things as mutual exclusion. In this example, mutual exclusion is clearly needed if a precaution like (1) is to ensure that initialization will only be performed once. But even then, to formalize the $ti < tj$ and $tj < ti$ argument of the previous page is no fun; as yet I have seen no more formal, yet "decent" argument. (A challenge when I have nothing else to do!) For practical reasons I would regard the argument on the previous page convincing enough. As yet the formal proofs tend to become hairy....

* * *

Finally I would like to congratulate you with your decision not to be content with the first solution and it justification. The experience that the next effort gives something much nicer is a very common one, but it is only believed by those who have had the experience themselves!

Yours ever,

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow

Burroughs
Plataanstraat 5
NUENEN - 4565, The Netherlands