meaningless. This is a fact which should not be used as an argument against the feature itself, if it is useful for other sound reasons.

(b) The **and** proposal essentially is equivalent to Anderson's **fork** and **join** statements, although it assumes an entirely different form. It is all too obvious that the **join** and **fork** concepts emerged from their author's close familiarity with computer organization and machine code programming, and it is perhaps unfortunate for him to have chosen ALGOL as a vehicle to illustrate his ideas. The very essential innovation provided by ALGOL was structure; for example, repeated or conditional execution of a statement can be expressed by preceding it by a for clause or an if clause, instead of through use of several labels and jumps referencing these labels. Therefore the **fork** and **join** statements using such labels are precisely contradictory to the fundamental principles on which ALGOL 60 is based.

In case 2, where parallel execution is compulsory, the sharing of variables causes some problems, and Mr. Anderson's **obtain** and **release** statements are obviously intended to be used in this connection. Strictly speaking, however, they are not necessary, as has been demonstrated by E. W. Dijkstra [3]. His "critical sections" supposedly contain statements where individual computers access shared facilities, and through which they can communicate with each other. Although his algorithm does not seem to be entirely satisfactory (cf. [4]), the problem of eliminating interference is solved. In practical multiprocessor computers, an interlocking facility will probably be provided as a hardware feature, and it might be desirable that it be properly reflected in a programming language. From such a feature one would expect correct handling of the queueing problem, in the sense that the order in which requests occur be also the order in which access is granted [4].

A possible solution to the problem of accessing common variables or devices consistent with the philosophy of ALGOL, would be the availability of a special procedure declaration (e.g., a procedure declaration preceded by the symbol **shared**) written in a block surrounding all blocks representing the parallel programs. Any one program calling upon such a procedure would invoke the built-in queueing algorithm for this procedure (or "critical section") granting execution of that procedure only if all previously queued processors have terminated execution of that procedure. An example is given below.

```
begin comment n programs are executed in parallel. Procedure
  "critical section" cannot be executed by more than one proc-
  essor at a time;
  shared procedure critical section; ··· ;
  begin comment program 1; ···
  end and
  begin comment program 2; ···
  end and ···
  begin comment program n; ···
  end
end
```


REFERENCES:

1. ANDERSON, J. P. *Comm. ACM 8*, 12 (Dec. 1965), 786.
2. VAN WIJNGAARDEN, A. Rep. MR76, Mathematisch Centrum, Amsterdam.
3. DIJKSTRA, E. W. *Comm. ACM 8*, 9 (Sept. 1965), 569.
4. KNUTH, D. E. *Comm. ACM 9*, 5 (May, 1966), 321–322, (letter to the editor).

NIKLAUS WIRTH
Computer Science Department
Stanford University
Stanford, California

[ED. NOTE. Programming language extensions quite similar to those above were independently suggested by J. M. Watt of the University of Liverpool in a letter received about the same time as the above letter. To avoid duplication, we will not print Professor Watt's letter here, except to mention that he suggests an even higher degree of parallelism so that "$a := b$ **and** $b := a$" is to denote interchange of the previous values of a and b.—G.S.]

Additional Comments on a Problem in Concurrent Programming Control

EDITOR:

Professor Dijkstra's ingenious construction [Solution of a Problem in Concurrent Programming Control, *Comm. ACM 8* (Sept. 1965), 569] is not quite a solution to a related problem almost identical to the problem he posed there, and Mr. Hyman's "simplification" for the case of two computers [*Comm. ACM 9* (Jan. 1965), 45] hardly works at all. I hope that by this letter I can save people some of the problems they would encounter if they were to use either of those methods.

It is easy to find a counterexample to Mr. Hyman's "solution." [I trust some readers may be able to understand his program although there are 15 syntactic ALGOL errors in 12 lines of program!] If initially $k = 0$ and $b[0] = b[1] =$ **true**, computer 1 may start the process by setting $b[1]$ **false**, subsequently finding $b[0]$ is **true**. Then computer 0 sets $b[0]$ **false** and finds $k = 0$, whereupon it starts to execute its critical section. But computer 1 now sets $k = 1$ and executes *its* critical section at the same time.

Professor Dijkstra's solution is harder to attack; however, there is a definite possibility that one or more of the computers which wants to execute its critical section may have to wait "until eternity" while other programs are executing theirs. In other words, although Dijkstra's algorithm ensures that all computers are not *simultaneously* blocked, it is still possible that an *individual* computer will be blocked. (He decided to allow this possibility in his statement of the problem since he was interested in cases where there is comparatively low average demand for the use of critical sections; but there are certainly many applications for which the possibility of individual blocking is unacceptable.) For example, suppose time passes in discrete intervals; this assumption is valid for many computer systems and it is convenient but not strictly necessary for this example. Assume computer 1 is looping, finding $b_k =$ **false** and $k \neq i$, and it is positioned at label $L13$ in Dijkstra's program at times 0, 10, 20, 30, ... It is quite possible for the other computers to set $b[k]$ and change k at other times so that computer 1 never breaks out of the loop. For example, if $k = N = 2$, computer 2 can set $b[2]$ **true** at times $10n+2$ and come back to $L20$ to set it **false** again at time $10n+9$, for arbitrarily many n.

I tried out over a dozen ways to solve this problem before I found what I believe is a correct solution. The program for the ith computer ($1 \leqq i \leqq N$), using the common store

integer array *control* [1:N], *integer* k

(initially zero) is the following:

```
begin integer j;
  L0: control [i] := 1;
  L1: for j := k step −1 until 1, N step −1 until 1 do
        begin if j = i then go to L2;
          if control [j] ≠ 0 then go to L1
        end;
  L2: control [i] := 2;
      for j := N step −1 until 1 do
        if (j ≠ i) ∧ (control [j] = 2) then go to L0;
  L3: k := i;
      critical section;
      k := if i = 1 then N else i − 1;
  L4: control [i] := 0;
  L5: remainder of cycle in which stopping is allowed;
      go to L0 end
```

To prove that this works, first observe that no two computers can be simultaneously positioned between their statements $L3$ and $L4$, for the same reason that this is true in Dijkstra's algorithm. Secondly, observe that the entire system cannot be blocked until all computers are done with their critical section computations; for if no computer after a certain point executes a critical section, the value of k will stay constant, and the first computer (in the cyclic ordering k, $k-1$, $\cdots$, 1, N, $N-1$, $\cdots$, $k+1$) which subsequently would wish to perform a critical section would meet no restraint.

Finally it is necessary to prove that no individual computer can become blocked. The proof of this is not trivial, for it can be shown that in unfavorable circumstances a computer positioned at $L1$ may have to wait as many as $2^{N-1}-1$ turns—while other computers do critical sections—before it can get into its own critical section. [For example, if $N = 4$ suppose computer 4 is at $L1$ and computers 1, 2, 3 are at $L2$. Then

(i) computer 1 goes (at high speed) from $L2$ to $L5$;
(ii) computer 2 goes from $L2$ to $L5$, then computer 1 goes from $L5$ to $L2$;
(iii) computer 1 goes from $L2$ to $L5$;
(iv) computer 3 goes from $L2$ to $L5$, then computer 1 goes from $L5$ to $L2$, then computer 2 goes from $L5$ to $L2$;
(v), (vi), (vii) like (i), (ii), (iii), respectively;

meanwhile computer 4 has been unfortunate enough to miss the momentary values of k which would enable it to get through to $L2$.

To prove that computer i_0 will ultimately execute its critical section after it reaches $L1$, note that since the system does not get completely blocked, i_0 can be blocked only if there is at least one other computer j_0 which does get to execute its critical section arbitrarily often. But every time j_0 gets through from $L0$ to $L5$, with *control* $[i_0] \neq 0$, the value of k it encounters at $L1$ must have been set by a computer k_0 which follows i_0 and precedes j_0 in the cyclic ordering N, $N-1$, $\cdots$, 2, 1, N. Therefore some k_0 which follows i_0 and precedes j_0 in the ordering must also execute a critical section arbitrarily often. This is a contradiction if we choose j_0 to be the first successor of i_0 having this property.

Lest someone write another letter just to give the special case of this algorithm when there are two computers, here is the program for computer i in the simple case when computer j is the only other computer present:

```
begin  L0: control [i] := 1;
       L1: if k = i then go to L2;
           if control [j] ≠ 0 then go to L1;
       L2: control [i] := 2;
           if control [j] = 2 then go to L0;
       L3: k := i; critical section; k := j;
       L4: control [i] := 0;
       L5: remainder; go to L0 end
```

When N is large or when it is variable, the above algorithm is not very efficient. Considerably more efficient methods can easily be designed if we modify the basic assumption that the only undividable operations of the computers are single reads or writes to a store. Suppose, for example, we have the common store

integer array $Q[0:N]$; **integer** T;

(initially zero) and assume that the three operations

procedure *add to queue* (i); $T := Q[T] := i$;
Boolean procedure *head of queue* (i); *head of queue* $:= (i = Q[0])$;
procedure *remove* (i); **if** $T = i$ **then** $Q[0] := T := 0$ **else** $Q[0] := Q[i]$;

are each indivisible, hardware operations. Then an efficient solution for the ith computer, $1 \leqq i \leqq N$, is:

```
begin  L0: add to queue (i);
       L1: if ¬ head of queue (i) then go to L1;
       L3: critical section;
       L4: remove (i);
       L5: remainder; go to L0 end
```

The loop at $L1$ can be handled by interrupts; so the processor can do other work while waiting in the queue. This method is "fairer" than the others, and it can be modified to work with priorities.

DONALD E. KNUTH
Mathematics Department
California Institute of Technology
Pasadena, California

Corrections to Numerical Data on Q-D Algorithm

EDITOR:

In the process of testing a polynomial-solving subroutine based on a scheme suggested by Zane C. Motteler [1], I used the tables of polynomials given in Watkins's and Henrici's article, "Finding Zeros of a Polynomial by the Q-D Algorithm" [2] as a source of test polynomials. A number of the entries proved to be in error, and I submit below correct versions of those entries in which I discovered errors.

TABLE I

Problem	*Degree*	*Coefficients*	*Zeros or Factors*
2	3	1, 0, 3, 1	−0.32219, 0.16109 ± *i*1.7544
3	3	1, 1.0004, −1.0002, −1.0006	−1.00000, 1.00010, −1.00050
4*	3	1, 3.00006, 3.000120, 1.00006	−1, −1, −1.00006
19†	6	1, 0, 0, 0, 0, 0, −1	1, −1, 0.50000±*i*0.86603, −0.50000±*i*0.86603
27	10	1, −2.5, −460.8, −9133.4, −50761.8, −88653.098, −53510.4, −37313, −197170, −364800, −198000	z^2-2z+2, $z^2+2z+2.2$, $z^2+20z+200$, $z+1$, $z+1.5$, $z+5$, $z-30$
29	13	1, 1289.01, 273087, 1612424.9, −1.488606E8, 1.1147382E9, −4.0019013E9, −2.49166E10, −1.0156643E10, 2.9101496E10, −2.954675E11, −1.1798729E12, −1.4200503E12, −5.5525336E11	z^2+3z+4, z^2-4z+8, $z^2-10z+64$, $z+1$, $z+2$, $z-16$, $z+32$, $z+256$, $z+1024$, $z+1.01$
30	15	1, 2, −334, −592, 36352, 60716, −1486310, −2191720, 23210431, 30731586, −169919436, −134375288, 1.634846E9, 2.1045721E9, −5.7149107E9, −6.2270208E9	z^2+4z+8, z^2-4z+9, $z+1$, $z-2$, $z+3$, $z+4$, $z-5$, $z-6$, $z+10$, $z-11$, $z-12$, $z+13$ $z+7$,

* Probably the polynomial intended here was $z^3 + 3.00006z^2 + 3.0001200005z + 1.0000600005$, which would give for roots −1, −1.00001, −1.00005. Truncation of the coefficients, however, changes the roots to those given in the corrected entry.

† I am guessing here that the authors intended to solve for the sixth roots of 1.

TABLE II

Problem	*Degree*	*Coefficients*	*Zeros or Factors*
4	7	3.8121, −74.419, −116.96, 4.2614, −9915.8, −0.07133, −0.04325, −4.3936	0.038108 ± *i*0.066060, −0.076218, 1.7322± *i*4.5447, −5.1782, 21.236
13	36	−9265.3, 6468.0, −42.015, 70.311, 3072.4, 2.9530, 5.6163 870.73, −7.9141, −74.110, −22.964, 9.2252, −2.4987, −39.063, 6.5810, −6,8461, −7.8867 −32.151, −34.637, 67.916, −390.57, 60.247, 265.74, −453.86, −7015.6, −309.67, −2.0574, −85.581, −99.394, −20.775 49.225, 3924.5, −0.083830, 73.941, 0.049060, 88.312, −993.56	0.81276 ± *i*0.061480, −0.58720 ± *i*0.48-197, 0.22083 ± *i*0.70000, −0.88460 ± *i*0.30152, 1.0530 ± *i*0.087703, −0.83729 ± *i*0.39630, −0.072582 ± *i*0.99088, 0.57888 ± *i*0.77201, −0.583111 ± *i*0.78522, 0.15853 ± *i*1.0003, 0.82646 ± *i*0.57735, −0.98285 ± *i*0.11248, 0.39262 ± *i*0.92500, 0.95385 ± *i*0.36071, −0.74665 ± *i*0.60412, −0.37723 ± *i*0.89807, 0.63308 ± *i*0.69314, −0.20948 ± *i*0.89996