Computer Systems
Robert L. Ashenhurst

Solution of a Problem in Concurrent Programming Control

E.W. Dijkstra

September, 1965
Volume 8, Number 9
pp. 569

This short paper marks the beginning of a long era of interest in expressing the synchronization of concurrent processors using ordinary programming languages. This paper considers the problem of implementing an indivisible operation by mutual exclusion of critical sections of code. Later papers by Dijkstra introduced the concepts of semaphores; one of the motivations of semaphores was reducing the complexity of the programming required to implement mutual exclusion. (A letter to the editor by Don Knuth, published in May 1966, points out that Dijkstra's solution is susceptible to "starvation" – meaning that a particular processor can fortuitously be forever blocked from entering the critical section. Knuth's starvation-free solution is even more complex than the solution here.)

–P. J.D.

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** $b, c[1{:}N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer $(1 \leq i \leq N)$ is:

```
"integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
     go to Li1
     end
       else
Li4: begin c[i] := false;
       for j := 1 step 1 until N do
         if j ≠ i and not c[j] then go to Li1
     end;
     critical section;
     c[i] := true; b[i] := true;
     remainder of the cycle in which stopping is allowed;
     go to Li0"
```

The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement *Li*4 without jumping back to *Li*1, i.e., finding all other c's **true** after having set its own c to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to *Li*1) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the kth computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in *Li*3 the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement *Li*4) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.

Letter to the Editor, Vol. 9, No. 1, January 1966, p. 45.

Comments on a Problem in Concurrent Programming Control

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control, *Comm ACM 8* (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

```
Boolean array b(0: 1) integer k, i, j,
comment This is the program for computer i, which may be
  either 0 or 1, computer j ≠ i is the other one, 1 or 0;
C0: b (i) := false;
C1: if k ≠ i then begin
C2:   if not b(j) then go to C2;
      else k := i; go to C1 end;
      else critical section;
      b(i) := true;
      remainder of program;
      go to C0;
      end
```

Mr. Dijkstra has come up with a clever solution to a really practical problem.

HARRIS HYMAN
Munitype
New York, New York

Letter to the Editor, Vol. 9, No. 5, May 1966, p. 321-322.

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 *L*13 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 *L*20 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 *L*3 and *L*4, 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 *L*1 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 *L*1 and computers 1, 2, 3 are at *L*2. Then

(i) computer 1 goes (at high speed) from *L*2 to *L*5;
(ii) computer 2 goes from *L*2 to *L*5, then computer 1 goes from *L*5 to *L*2;
(iii) computer 1 goes from *L*2 to *L*5,
(iv) computer 3 goes from *L*2 to *L*5, then computer 1 goes from *L*5 to *L*2, then computer 2 goes from *L*5 to *L*2;
(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 *L*2.

To prove that computer i_0 will ultimately execute its critical section after it reaches *L*1, 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 *L*0 to *L*5, with *control* $[i_0] \neq 0$, the value of k it encounters at *L*1 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 *L*1 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

Letter to the Editor, Vol. 11, No. 1, January 1968, p. 55-56.

Additional Comments on a Problem in Concurrent Programming Control

EDITOR:

In D. E. Knuth's solution [*Comm. ACM 8*, 5 (May, 1966), 321-322, Letter to the Editor] of Dijkstra's problem [*Comm. ACM 8*, 9 (Sept. 1965), 569] it is not quite easy to check that any computer waiting for its critical section has to wait at most 2^{N-1} turns (the word "turn" refers to a computer performing its critical section). It occurred to me that by a small change in his program that number can be reduced to $\frac{1}{2}N(N-1)$, with the extra advantage that for this new program it is easier to see why and how it works. The change consists of replacing

```
L3: k := i;
    critical section;
    k := if i=1 then N else i−1;
```

by

```
L3: critical section;
    if control [k] = 0 ∨ k=i then k := if k = 1 then N else k−1;
```

and requiring that the initial value of k is one of the numbers $1, \cdots, N$, instead of 0.

With these alterations we find:

(i) If at a certain moment k has a value i, and if control $[i] \neq 0$, then k does not change its value before computer i performs its critical section.

(ii) In a time interval where k is constant, no computer can pass its critical section twice. Assuming computer j passes twice, we have $j \neq k$ and control $[k] \neq 0$ (otherwise k would have changed the first time); computer k does not pass its critical section before j does (for otherwise the value of k would change before j gets its second turn); hence control $[k] \neq 0$ all the time between the two turns of j, and this means that j cannot get to *L*2 after its first turn.

From (i) and (ii), it follows that if computer i has control $[i] \neq 0$, then it has to wait at most $N(N-1)$ turns. The actual maximum is $\frac{1}{2}N(N-1)$, however. This we prove for $i = 1$.

(iii) If j has one of the values $2, \cdots, N$, then the following holds. In a time interval throughout which control $[1] \neq 0$ and $j \geq k \geq 1$, computer j can pass its critical section at most once. For, after its first passage we cannot have $k = j$, (even if $j = N$ the value of k cannot jump from 1 to N under these circumstances) and from that moment on we have $j > k \geq 1$, which implies that j cannot get to *L*2 before control $[1] = 0$.

From (i), (ii), and (iii), it follows that in a time interval where control $[1] \neq 0$, computer j can have at most $N-j+1$ turns ($2 \leq j \leq N$). Hence computer 1 has to wait at most $\sum_{j=2}^{N} (N-j+1) = \frac{1}{2}N(N-1)$ turns. It is not difficult to show that this waiting period can indeed occur.

N. G. DE BRUIJN
Technological University
Eindhoven, The Netherlands