Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra

Technological University, Eindhoven, The Netherlands

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 only one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given 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 indivisible 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 solution.

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, (1 ≤ i ≤ N), integer k"

The integer k will satisfy 1 ≤ k ≤ N, b[k] and k 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 ≤ i ≤ N) is:

```
integer j;
L0: b[i] := false;
L1: if k ≠ i then
    L2: begin c[i] := true;
    L3: if k(i) then k := j;
        go to L1
    else
    L4: begin c[i] := false;
        for j = 1 step 1 until N do
            if j ≠ i and not c[j] then go to L1
        end
    critical section:
    c[i] := true;
    b[i] := true;
    remainder of the cycle in which stopping is allowed;
        go to L0
```

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 only the way to enter its critical section is the performance of the compound statement L0 without jumping back to L1, 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 L0) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the ith computer is not among the looping ones, b[i] will be true and the looping ones will all find k ≠ i.

As a result one more of them will find in L3 the Boolean b[i] 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 L4) 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 ≠ i. And this, the author believes, completes the proof.
Additional Comments on a Problem in Concurrent Programming Control

Professor Djikstra's ingenious construction (Solution of a Problem in Concurrent Programming Control, Comm. ACM 8 (Sept. 1965), p. 396) is no solution to a related problem almost identical to the problem he posed there, and Mr. Hyman's "simplification"1 for the case of two computers (Comment. ACM 8 (Dec. 1965), p. 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."

[The text is not fully legible due to the image quality, but it appears to discuss a computer scheduling problem and provides a counterexample to a proposed solution.]

Professor Djikstra's solution is easier to attack, however, there is a definite possibility that one of the computers which wants to execute its critical section may have to "wastefully" execute other programs before executing its critical section. In other words, although Djikstra'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 a minimal number of processes blocked for the use of critical sections; but there are many applications for which the possibility of individual blocking is unacceptable.) For example, suppose times are 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 hogging, finding $b_1 = \text{false}$ and $k_1 = 1$, and it is precluded at label L13 in Djikstra's program at times 0, 30, 60, ... . It is quite possible for the other computers to set $b_i$ and change it at other times so that computer 1 never breaks out of its loop. For example, if $k_1 = 2, \text{computer } 1 \text{ can set } b_i \text{ to } \text{true} \text{ at times 90, 120, and go back to L12 to set } b_1 \text{ again at time 1000, for arbitrarily many } n$.

Last summer I found out a value smaller than the one I found before I found what I believe is a correct solution. The program for the $i$th computer ($1 \leq i \leq N$), using the common store

```
integer array control [i:N], integer k
```

(instead of zero) in the following:

begin integer j; integer k;

L0: control [j] = 1; L1: if $k = j$ step -1 until 1, $N$ step -1 until 0 do begin if $k = j$ then go to L3; if control [j] = 0 then go to L1 end;

L2: control [j] = 2; for $j = N$ step -1 until 1 do if $j < i$ then control [j] = 2 then go to L0;

L3: critical section;

$k = k + 1$ then $N$ else $j = 1$;

L4: control [j] = 0;

L5: remainder of code 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 Djikstra's algorithm. Secondly, observe that the entire system cannot be blocked until all computers are done with their critical section computations; if for 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 1, 2, ... , $N$) will output a value as $2^{N-1}$ turns—while other computers do critical sections before it can get into its own section. If $N > 1$ suppose computer 1 is as L1 and computer 2, 3 are at L2. Then

(i) computer 1 goes (at high speed) from L2 to L3;

(ii) computer 2 goes from L2 to L3, then computer 1 goes from L5 to L2;

(iii) computer 1 goes from L2 to L3, then computer 1 goes from L5 to L2, then computer 2 goes from L5 to L3; and

(iv) (i) (ii) (iii) (iv), respectively;

meanwhile computer 1 has been unfortunate enough to miss the momentary value of $k$ which would enable it (get through to L2).

To prove that computer 1 will ultimately execute its critical section after it reaches L1, note that since the system does not get completely blocked, computer 1 can be blocked only if there is another computer other than 1 which does not get to execute its critical section arbitrarily often. But every time $j$ gets through from L2 to L3, control [j] at 0, the value of $k$ it encounters at L1 must have been set by a computer other than 1 which sets $k = k + 1$ and precedes $j_0$ to the cyclic ordering $N, N-1, ... , 1, N$. Therefore some other computer which follows $j_0$ and precedes $j$ is the only one who must also execute the critical section arbitrarily often. This is a contradiction if we choose $j_0$ as the first successor of $j$ having this property.

Last spring I found out a smaller value of $k$ for the special case of this algorithm when there are two computers, here is the program for computer 1 in the simple case when computer 2 is the only other computer:

begin L0: control [j] = 1; L1: if $k = j$ then go to L2; if control [j] = 0 then go to L1;

L2: control [j] = 2; if control [j] = 2 then go to L1;

L3: critical section;

$k = k + 1$ then $N$ else $j = 1$;

L4: control [j] = 0;

L5: remainder of code in which stopping is allowed; go to L0 end.

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

```
integer array control [0:N], integer T;
```

initially zero and assume that the three operations

- procedure add to queue (i); $T := Q[i]$;
- procedure head of queue (i); head of queue $= (i + Q[i])$

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

begin L0: add to queue (i); L1: if head of queue (i) then go to L7;

L3: critical section;

L4: remove (i)

L5: remainder; go to L0 end.

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

DOROTHY A. KAYN Mathematics Department California Institute of Technology Pasadena, California

---

Additional Comments on a Problem in Concurrent Programming Control

Editor:

In the D. E. Knuth's solution (Comm. ACM 5, May 1966, p. 391) to the letter to the Editor of Djikstra's problem (Comm. ACM 5, Sept. 1965, p. 396) 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 $N(2N-1)$, with the extra advantage that for the very small program it is easier to see why and how it works. The change consists of replacing

L3: $k = 1$

by

L3: if control [k] = 0 then k = 1 else

and requiring that the initial value of $k$ is one of the numbers 1, 2, ..., $N$, instead of 0.

With these alterations we find:

(i) if at a certain moment $k$ has a value other than 1, 2, ... , $N$, then it does not change its value before computer i performs its critical section.

(ii) if in a time interval where $k$ is constant, no computer can pass its critical section twice. Assuming computer $j$ passes twice, we have $j = 2$ and control [j] = 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 $j$ goes to the second turn); hence control [k] = 0 at all the time between the two turns of $j$, and this means that $j$ cannot get to L2 after its first turn.

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

(iii) If $i$ has one of the values 1, ... , $N$, then the following holds. In a time interval throughout which control [i] = 0 and $j = 2$, $j$ = 3, $j$ = 4, computer $j$ can pass its critical section at most once. For, after its first passage we cannot have $j = k$ (even if $j = k$ the value of $k$ cannot jump from $k$ to $N$ under these circumstances) and from that moment on we have $j > k = 2$, which implies that $j$ cannot get to L2 before control [i] = 0.

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