A NEW SOLUTION TO THE CRITICAL SECTION PROBLEM[†]

Howard P. Katseff
Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California
Berkeley, California 94720

(Extended Abstract)

Introduction

A classical problem in concurrent program control is to provide a mechanism whereby several processes running concurrently can gain exclusive control of a resource. For each process, the section of its program in which it accesses the resource is called its ***critical section,*** and the problem is called the ***critical section problem.*** A solution to the critical section problem guarantees that no more than one process can be in its critical section at any time. The solution presented here improves on previous solutions by allowing processes to enter their critical sections on a first-come first-served basis.

The problem was originally stated and solved by Dijkstra in 1965. In his, as well as in subsequent solutions it is assumed that the only indivisible operations are single reads and writes to variables and that nothing can be said about the relative speeds of the different processors. Unfortunately, his solution has the property that in some cases a process may never be allowed to enter its critical section. A later solution by Knuth (1966) has corrected this problem.

More recent solutions have been formulated which function under the more demanding requirements of a distributed system. See, for example, papers by Rivest and Pratt (1976) and Peterson and Fischer (1977). Here, processes are allowed to die at any time, yet the solution must still function for the remaining processes. Note however that processes must indicate their death in a prescribed fashion, say by setting some variable to zero. These solutions also do not require global variables into which more than one process can write. All communication is done by variables which are local to each process in the sense that they can only be written by the process they are local to but can be read by all processes.

Ideally, the critical section solution should serve processes in a fair manner. In Knuth's solution, with an n process system, a process may have to wait for other processes to enter their critical sections $2^{n-1}-1$ times. Eisenberg and McGuire (1972) improved on this bound by providing a solution in which a process will never wait more than $n-1$ turns.

A better approach is to serve processes on a first-come first-served basis. Each process executes a section of code, called the ***protocol,*** before entering its critical section. There is a statement in the protocol called the ***doorway*** with the property that whenever a process i executes this statement (passes through the doorway) any process which later begins its protocol will execute its critical section after process i does.

A recent solution by Peterson and Fischer (1977) meets these conditions, except that a process may have to wait indefinitely before entering its doorway. In Lamport's (1974) solution processes are served on a first-come first-served basis, but a process which repeatedly dies and reawakens may inhibit other processes from ever entering their critical sections. Furthermore, his solution requires variables which assume arbitrarily large values. Our solution serves processes in a first-come first-served manner without any of the above problems.

The Solution

We first describe a simple algorithm which, while it does not correctly solve the critical section problem, helps introduce the key ideas of the actual solution. We use the variable n to indicate the number of processes in the system. Each process has a unique number between 1 and n. The protocol for each process is identical except that the variable i contains its process number. A process indicates its death by setting all variables used in this algorithm to zero. All variables are also initially set to zero.

We now describe a mechanism, due to Peterson and Fischer (1977), which ensures that processes are served on a first-come first-served basis. When a process wishes to enter its critical section, it first examines the other processes and makes a list of those which have already passed through their doorways. This list is called a ***behind-of list.*** Before entering its critical

[†]This work was supported by National Science Foundation Grant MCS75-21760-A01.

section, a process must wait until each process in its behind-of list either completes its critical section or dies.

Consider the following attempt at a solution to the critical section problem:

```
1. Build a behind-of list consisting of those
     processes $j$ with $waiting_j = 1$.

2. $waiting_i := 1$

3. repeat
       Delete from the behind-of list those
       processes $j$ with $waiting_j = 0$.
   until the behind-of list is empty

4. CRITICAL SECTION

5. $waiting_i := 0$
```

In this algorithm, step 2 is the doorway. After process i passes through its doorway, all other processes will place it on their behind-of lists and will not execute their critical sections until process i does. However, this algorithm fails to be a solution to the critical section problem in two ways. First, it is possible for more than one process to be in its critical section at the same time. Second, it is possible for a process to wait forever to enter its critical section.

How can two processes be in their critical sections at the same time? Suppose that we have a two process system and that both processes simultaneously begin their protocols and execute step 1. They will both have empty behind-of lists and thus can both proceed through steps 2 and 3 and enter their critical sections.

This problem is corrected by placing the following code, adapted from Eisenberg and McGuire (1972), between steps 3 and 4.

```
L0: $control_i := 1$
L1: for $j := 1$ step 1 until $i-1$ do
      if $control_j \neq 0$ then goto L1
L2: $control_i := 2$
      for $j := 1$ step 1 until $n$ do
          if $j \neq i$ and $control_j = 2$ then goto L0
```

The statement

$control_i := 0$

should also be added to step 5.

The three lines of code starting at L2 guarantee that there is never more than one process in its critical section at any time. For example, consider the case where two processes simultaneously execute the statement at L2. Each will then notice that the other has its element of *control* set to 2 and will thus go back to L0. Now, only the lower numbered process will get past the loop at L1 and go on through the second loop to its critical section.

We now look at the other problem with this algorithm: that it is possible for some process to wait forever to enter its critical section. We will show that it is possible to have a ***cycle*** of behind-of lists. This refers to a set of processes $i_1, i_2, ..., i_m$ with the property that i_{k+1} is in process i_k's behind-of list for $k < m$ and i_1 is in process i_m's behind-of list. If a cycle occurs, then all processes in the cycle will get stuck at step 3 and wait forever.

Suppose that we have a two process system in the following situation: process 1 is in the loop at step 3 with process 2 in its behind-of list and process 2 is executing its critical section. Now, suppose that process 2 quickly exits its critical section and then reenters its protocol before process 1 has a chance to notice that process 2 has completed its critical section. Each process will be in the other's behind-of list, providing the desired example.

To correct this problem, we introduce additional variables to determine which process has last begun its protocol. For process i we have the n variables:

$count_i[1], count_i[2], ..., count_i[n]$.

While process i is in its protocol, these variables take on values between 1 and 3. When it is dead or not contending for its critical section, they are all zero.

Values are assigned to these variables by inserting the following loop as the new first statement in the algorithm:

```
for $j := 1$ step 1 until $n$ do
    $count_i[j] := (count_j[i]$ modulo 3) + 1
```

Now, if $count_j[i] = (count_i[j] \bmod 3) + 1$ then process j entered its protocol after process i did. The *count*'s are used in step 3. Process i removes j from its behind-of list if $count_j[i] = (count_i[j] \bmod 3) + 1$ because this means that j entered its protocol after i and therefore does not belong on i's behind-of list.

The Algorithm

We have now explained all significant portions of our solution to the critical section problem. In detail, the algorithm is:

```
1. for $j$ := 1 step 1 until $n$ do
       $count_i[j] := (count_j[i] \text{ modulo } 3) + 1$

2. Build a behind-of list consisting of those
     processes $j$ with $waiting_j = 1$.

3. $waiting_i := 1$

4. repeat
       Delete from the behind-of list those
       processes $j$ with $count_j[i] = 0$ or
       $count_j[i] = (count_i[j] \text{ modulo } 3) + 1$.
   until the behind-of list is empty

5. L0: $control_i := 1$
   L1: for $j$ := 1 step 1 until $i-1$ do
           if $control_j \neq 0$ then goto L1
   L2: $control_i := 2$
       for $j$ := 1 step 1 until $n$ do
           if $j \neq i$ and $control_j = 2$ then goto L0

6. CRITICAL SECTION

7. $control_i := waiting_i := 0$
   for $j$ := 1 step 1 until $n$ do
       $count_i[j] := 0$
```

Whenever a process i dies, the variables $waiting_i$, $control_i$ and $count_i[1]$, $count_i[2]$, ..., $count_i[n]$ are all set to zero. They also have zero as their initial values. A proof that the algorithm solves the critical section problem and serves processes in a first-come first-served manner is given in Katseff (1978).

Acknowledgments

The author would like to thank Manuel Blum, Leslie Lamport and Gary Peterson, who all helped debug the algorithm. I am also grateful to Brenda Baker for her help in revising the manuscript.

References

Dijkstra, E. W. (1965), "Solution of a Problem In Concurrent Programming Control," *Comm. ACM 8,* 569.

Eisenberg, M.A. and McGuire, M.R. (1972), "Further Comments on Dijkstra's Concurrent Programming Control Problem," *Comm. ACM 15,* 999.

Katseff, H. P. (1978), "A Solution to the Critical Section Problem with a Totally Wait-free FIFO Doorway," Internal Memorandum, Computer Science Division, University of California, Berkeley.

Knuth, D.E. (1966), "Additional Comments on a Problem in Concurrent Programming Control," *Comm. ACM 9,* 321.

Lamport, L. (1974), "A New Solution of Dijkstra's Concurrent Programming Problem," *Comm. ACM 17,* 453.

Peterson, G.A. and Fischer, M.J. (1977), "Economical Solutions for the Critical Section Problem in a Distributed System," *Proc. ACM Symp. Thy. Comp. 9,* 91–97.

Rivest, R.L. and Pratt, V.R. (1976), "The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report," *Proc. IEEE Symp. Found. Comp. Sci. 17,* 1–8.