# MYTHS ABOUT THE MUTUAL EXCLUSION PROBLEM

G.L. PETERSON
*Department of Computer Science, University of Rochester, Rochester, NY 14627, U.S.A.*

Received 13 January 1981; revised version received 30 March 1981

Parallelism, mutual exclusion

Recently in these pages appeared a report by Doran and Thomas [2] which gave partially simplified versions of Dekker-like solutions to the two process mutual exclusion problem with busy-waiting. This report presents a truly simple solution to the problem and attempts in a small way to dispel some myths that seem to have arisen concerning the problem.

Briefly, the *mutual exclusion problem* for two processes is to find sections of code (*trying protocol, exit protocol*) for each of two asynchronous processes to use when trying to enter and upon exiting their designated critical sections. The protocols must preserve mutual exclusion and not have deadlock or lockout. *Mutual exclusion* means that both processes can never be in their critical sections at the same time. No *deadlock* or *lockout* means that no process waits forever inside a protocol. More formal definitions can be found in [5] and elsewhere.

The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are slight improvements which eliminate the 'loop inside a loop' structure of the previously published solutions. The solution presented here has an extremely simple structure and, as shown later, is easy to prove correct.

The protocols of $P_1$ and $P_2$ are given in Fig. 1. Q1 and Q2 are initially *false* and TURN may start as either 1 or 2. (The busy wait loop '*wait until* Boolean' is just another way of saying "*repeat*/* empty statement/* *until* Boolean". The Boolean formula is *not* evaluated atomically.)

As can be seen, the algorithm has a very simple structure. This results in an easy proof of correctness. First, neither process can be locked out. Consider $P_1$, it has only one wait loop, and assume it can be forced to remain there forever. After a finite amount of time, $P_2$ will be doing one of three general things: not trying to enter, waiting in its protocol, or repeatedly cycling through its protocols. In the first case, $P_1$ notes that Q2 is *false* and proceeds. The second case is impossible due to TURN being either 1 or 2; and one of the processes will proceed. In the third case $P_2$ will quickly set TURN to 2 and never change it back to 1, allowing $P_1$ to proceed.

If mutual exclusion were not preserved and both processes could somehow end up in their critical sections at the same time, then we have Q1 = Q2 = *true*. Their tests in their wait loops just prior to entering their critical sections at this point could not have been at approximately the same time as TURN would have been favorable to only one of the processes and the other part of the test would have failed for both. This

```
/*trying protocol for P1*/
Q1 := true;
TURN := 1;
wait until not Q2 or TURN = 2;
Critical Section;
/*exit protocol for P1*/
Q1 := false.
```

```
/*trying protocol for P2*/
Q2 := true;
TURN := 2;
wait until not Q1 or TURN = 1;
Critical Section;
/*exit protocol for P2*/
Q2 := false.
```

Fig. 1. A simple solution.

implies that one process first passed its test, and the second did one or more assignments before passing its test by seeing TURN favorable to itself. However, the last assignment before testing sets TURN to an unfavorable value, the test is doomed to fail, and mutual exclusion is preserved.

Since the more complex algorithms naturally require more complex proofs, one wonders whether the prevalent attitude on 'formal' correctness arguments is based on poorly structured algorithms. Perhaps good parallel algorithms are not really all that hard to understand. In any case, this solution puts an end to the myth that the two process mutual exclusion problem requires complex solutions with complex proofs. (Dijkstra has recently devised a more formal proof of mutual exclusion for this algorithm [7] which, to this author, seems unnaturally complex for such a simple algorithm.)

This algorithm does not appear 'out of nowhere', but is in fact easily derivable from simple forms. Consider the two primitive algorithms in Fig. 2, both slight modifications of ones appearing in [1]. Note that the first has no exit protocols! Both algorithms preserve mutual exclusion but both have deadlock. The first only when one process does not cyclically try and the second only when they both are trying. The waiting problems are disjoint and the correct algorithm is a simple combination of the two. The myth of difficulty of derivation is laid to rest.

```
TURN := 1;                    TURN := 2;
wait until TURN = 2;          wait until TURN = 1;
Critical Section              Critical Section

Q1 := true;                   Q2 := true;
wait until not Q2;            wait until not Q1;
Critical Section;             Critical Section;
Q1 := false.                  Q2 := false.
```

Fig. 2. Two primitive solutions.

Another possible myth is that Dekker's solution can be trivially modified to solve the n process case. The algorithms known to the author actually require major changes in form that result in entirely new algorithms, even when n is two. The solution given here does have a generalization to n processes. This algorithm (based on a three process solution by L. Hrechanyk) is given in Fig. 3. The two process solution is used repeatedly in n – 1 levels to eliminate at least one process per level until only one remains. The shared arrays Q[1 ··· n] and TURN[1 ··· n – 1] are initially 0 and 1, respectively. The variables i, n and j are local to the process with i containing the process number and n the total number of processes. The correctness proof of this solution is a straightforward generalization of the two process proof and is left to the reader. However the algorithm requires 2n – 1 shared variables of size n. Algorithms that need far fewer variables while satisfying more constraints are well known [4–6].

```
/*protocols for P_i*/
for j := 1 to n - 1 do
begin
  Q[i] := j;
  TURN[j] := i;
  wait until (∀k ≠ i, Q[k] < j) or TURN[j] ≠ i
end;
Critical Section;
Q[i] := 0
```

Fig. 3. Simple n process solution.

## References

[1] E.W. Dijkstra, Co-operating sequential processes, in: F. Genuys, Ed., Programming Languages (Academic Press, New York, 1968) 43–112.

[2] R.W. Doran and L.K. Thomas, Variants of the software to mutual exclusion, Information Processing Lett. 10 (4/5) (1980) 206–208.

[3] R.C. Holt, G.S. Graham, E.D. Lazowska and M.A. Scott, Structured Concurrent Programming with Operating Systems Applications (Addison-Wesley, Reading, MA, 1978).

[4] L. Lamport, The mutual exclusion problem, SRI International (1980).

[5] G.L. Peterson, Concurrency and complexity, TR59, Dept. of Computer Science, Univ. of Rochester (1979).

[6] G.L. Peterson, A new solution to Lamport's concurrent programming problem using small shared variables, Dept. of Computer Science, Univ. of Rochester (1980).

[7] E.W. Dijkstra, An assertional proof of a program by G.L. Peterson, EWD 779, Burroughs Corp. (1981).