**On Variations of Peterson's Mutual Exclusion Algorithm**
by

Ross Lee Graham
Kazakhstan Institute of Management Economics and Strategic Research
ross@rosslg.com
and
Ali Nademi
Mid Sweden University ITM
seyed_@hotmail.com

**Abstract**
*Mutual exclusion algorithms used for concurrent processes are designed to permit only exclusive access to shared critical sections. In 1981 the most concise version presented for two concurrent processes was Peterson's Algorithm. Peterson used the OR operator in the decision control. Tanenbaum uses a claimed version of Peterson's Algorithm that uses the AND operator in the decision control and there is no resetting of the flags. We show that this AND version leads to a trivialization of Peterson's original form. The first cycle, which looks interleafed, reverts to batch processing. Since batch processing is in serial order, this eliminates the need for a mutual exclusion algorithm designed for concurrent processes. Using Peterson's original OR operator and resetting the flags as he does, every run is interleafed. Furthermore, as should be expected, a DeMorgan on the Peterson control operator yields an AND version that sustains the interleafing identical to Peterson's original form. However, this form is clearly not simpler than the original OR form. It requires three additional NOT operators and the flags must still be reset.*

**Introduction**
We distinguish two general modes for obtaining mutual exclusion of concurrent processes. One mode concerns control of the processing threads. The second mode controls the data used by the processes.

Mutual exclusion using data control is found in several well-known data locking algorithms used in databases. Peterson's algorithm, and the main concern for this paper, addresses the traffic control problem of threads for mutual exclusion from critical sections in concurrent processing.

It is often difficult to design correct coordination schemes for asynchronous activities, such as process control, traffic control, stock control, banking applications, centralized computer services, managing information flow in large organizations, and etc. When we deal with information systems with several processes running concurrently, the mutual exclusion problem arises.

**Peterson's Algorithm**
A thread solution for the mutual exclusion problem was presented by Dekker in 1962. In 1981, G. L. Peterson presented a remarkable and quite elegant reduction that requires only four lines of instructions. The underpinning logic for his solution follows Dekker's intent, to interleaf the concurrent threads such that a mutual exclusion is sustained for all critical sections. In both algorithms, both the `turn` variable and status flags (here `Q1` and `Q2`) are used. The Exhibit below shows Peterson's Algorithm as he presented it in "*Myths about the Mutual Exclusion Problem.*" In the beginning, the flags `Q1` and `Q2` can have the value *false*, and the `turn` variable can have either the value of `1` or of `2`.

```
/*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
```

**Exhibit:** Peterson's Algorithm

We address first how Dekker's Algorithm can be so reduced and still be correct. The first issues that we inspect are whether any concurrent process controlled by Peterson's version can end up in a deadlock or lockout (starvation). We can show that neither process can be locked out [1]. To prove this we can assume that `P1` is in its wait loop. Would `P1` be stuck there? `P2` in the meantime has only three alternatives:

1. Not try to enter the critical section
2. Wait in its loop
3. Constantly cycle through its protocol

In the first case it is easy to show that if `P2` does not need to enter the critical section that the flag `Q2` remains *false* so that `P1` finds `Q2` *false* and exits the waiting loop. Therefore, the second case in the list can never happen; neither of them end up in an infinite waiting state because the `turn` variable can only belong to one of the processes, it is a `1` or a `2` (showing which process it belongs to). Finally in the third and last condition, `P2` sets the `turn` variable to `2` and as the algorithm shows, `P2` never changes the `turn` variable back to `1` again. Thereby `P1` is never stuck in its waiting loop and a deadlock or lockout is impossible.

The second issue is whether it can be guaranteed that the mutual exclusion is preserved. Despite all odds if both processes find themselves in the critical section simultaneously, then both flags `Q1` and `Q2` would be equal to *true*. But in the waiting loop another condition must be tested, namely, the `turn` variable. The `turn` variable can only take the value of one of the processes and thereby the testing part in the other process waiting-loop will fail. The result is that only one process can enter the critical section first and when it is finished the second process enters. Mutual exclusion is thereby guaranteed [1].

A third issue is whether interleafing is always sustained for the two processes, which is a defined condition for using a mutual exclusion algorithm. Or whether the algorithm degenerates to batch processing (processes running in serial order), which excludes the necessity of a mutual exclusion algorithm.

**Verification of Peterson's Algorithm with SPIN**
We used SPIN to verify Peterson's mutual exclusion algorithm. By inspecting the verification outputs, the simple recursion of the interleafing of concurrent processes was rendered visible. This also renders obvious a simple induction proof for the algorithm. SPIN shows that no errors were counted in the recursion. This indicates that assertion violations are impossible in Peterson's Algorithm.

We set our first simulation to run 500 cycles. However, the recursion became visible in the second cycle and shows no changes in the interleafing in any of the subsequent cycles. By inspecting the recursion pattern it is evident that the simulator creates 2 processes, SPIN named *proc 0* and *proc 1* (referenced in our text as `P1` and `P2`) and that both of these processes are interleafed throughout the run, sustaining mutual exclusion for the critical sections. That Peterson's Algorithm solves the mutual exclusion problem for two processes running concurrently, i.e., sustaining the interleafing, becomes an important point when considering some variants that are claimed as improvements on his original form.

Peterson therefore succeeded in developing a simpler solution to the mutual exclusion problem, and his solution can replace the more complex solutions that not only require more complex proofs, but are more difficult to implement.

**SPIN Verification of the dual variant of Peterson's Algorithm**
Is it possible to maintain the same interleafing as Peterson's original form using variations of it?

Starting with Peterson's control condition where we find the OR operator in the form ($p \vee q$), which underpins the key line known as the "waiting" line:

```
(flag[j] == false || turn != i) ->
/* wait until true */
```

By separating this line into its two component statements we equate these to $p$ and $q$ such that:
$p$ `flag[j] == false`
$q$ `turn != i`

Therefore, with a DeMorgan we obtain

$$\sim (\sim p \wedge \sim q),$$

which we translate as follows:

```
(flag [j] == false) goes to
(flag [j] == true)

(turn != i) goes to
(turn ==i)

(flag [j] == false || turn != i) goes to
((flag [j] == false || turn != i) ==
(false))

(flag [j] == true && turn == i) goes to
((flag [j] == true && turn == i) == false)
```

Here we verified with SPIN that this AND operator variant to Peterson's algorithm is also assertion violation impossible. There are no errors counted, same as the original version.

Our first simulation outputs were set to 500 cycles. Again, only a few were actually required to render visible the sustained recursion of interleafing. We clearly obtain that both processes are executed in an interleafed manner. Not many cycles are required to see a sustained recursion where no flag is blocked and both flags reset. However, this form

is not simpler than Peterson's original version since it requires the addition of three negations.

**Verification attempts at other claimed variants of Peterson's Algorithm with SPIN**

During this research we also came across another attempt on Peterson's Algorithm in an alleged variant form. It is further claimed to be an improvement. In many sources [25] [2] [15] [22] [23] [24] [9] we have seen that the OR operator is replaced by the AND operator.

Andrew S. Tanenbaum, professor of computer science at Vrije Universiteit in Amsterdam, Netherlands, in his popular book "Modern Operating Systems" [9] has also applied one of these alleged AND forms of Peterson's Algorithm as follows:

```
#define FALSE 0
#define TRUE  1
#define N             2
/*number of processes*/

int turn;
/*whose turn is it?*/
int interested[N];                    /*all
values initially 0(FALSE)*/

void enter_region(int process);
/*process is 0 or 1*/
{
       int other;
/*number of the other process*/

       other = 1-process;             /*the
opposite of process*/
       interested[process] = TRUE;
/*show that you are interested*/
       turn = process;
/*set flag*/
       while(turn     ==     process     &&
interested[other] == TRUE)    /*null
statement*/
}

void leave_region(int process)
/*process: who is leaving*/
{
       interested[process]= FALSE;
/*indicate    departure    from    critical
section*/
}
```

Tanenbaum references the original version of Peterson's Algorithm [1] but there is no indication of why this AND version was used.

Professor Jim Mooney from Ohio State University in [15] declares that problems would occur if we replace the AND operator by the OR operator. He states that "*since a process will be forced to wait if either condition is true, it will wait if it does not have the turn, even if the other process is not busy. Moreover, since it is possible for both processes to set their flags to busy, they may both wait even though one if them has the turn. Progress is violated, and deadlock is possible*". Our results show that his statement is not correct; deadlock is not possible in the OR-variant.

We created the PROMELA version of the alleged improvement of Peterson's Algorithm based on the AND operator. The code is almost the same as the original version.

In the original version of Peterson's Algorithm we have the OR operator as follows:

```
(flag[j] == false || turn != i) ->
/* wait until true */
```

The alleged improvement on Peterson's Algorithm based on the AND operator gives:

```
(flag[j] == true && turn == j) ->
/* wait until true */
```

By inspecting the lines in the SPIN outcome, we observe that no errors are counted. This indicates that assertion violations are also impossible in this alleged variant of Petersons's Algorithm.

```
State-vector 20 byte, depth
   reached 19, errors: 0
```

Why did Peterson in [1] used the OR operator when we clearly have obtained that the AND operator in the alleged variant works without any further error or problems? Finally, what was the motive to change the OR to the AND operator? We did more tests, to find the answers. This led us to understand why the mistaken AND form has passed scrutiny and has been accepted as if correct.

For the first test we ran the simulation for 500 steps. An inspection of the outputs shows again that the simulator creates 2 processes, named *proc 0* and *proc 1*. But for the alleged variants we can see that both of these processes are just interleafed in the first cycle (when the flags were set for the run), afterwards these processes ran in serial order (no interleafing, i.e., batch processing) and the flags were never reset. These runs are *not* interleafed. This is what led us to call this variant 'alleged'. This alleged variant of Petersons's Algorithm does not solve the interleaf requirement for concurrency of two processes.

Testing 140 cycles gave us more than sufficient data for revealing the recursion. As it shows by the *cnt* variable we can see that this algorithm does not allow these two processes to enter the critical section at the same time, but once again the processes are put in serial order and this is not a

solution that interleafs process segments.

By adapting De Morgan's law we logically show that the simplest acceptable AND variant is the dual variant of the original OR version. Running the DeMorgan variant by SPIN we received exactly the same result as the original OR version.

**Conclusion**

Based on this work we conclude that the simplest AND variant of Peterson's Algorithm that works is the DeMorgan on the original OR version. We observe also that it is not simpler than his original version. Furthermore, the AND version promoted by Mooney [25] and others as a simplification of Peterson's Algorithm is now proven to be a false claim and needs to be systematically eliminated from texts and papers that use it.

**Future work**

One of the issues to consider for mutual exclusion algorithms is their scalability. Peterson showed in his first paper [1] an n-process version of his Algorithm developed by his colleague at Rochester, Lydia Hrechanyk (now deceased).

```
/* protocols for P_I*/
for j = 1 to n-1 do
{
   Q[i] = j;
   TURN[j] = i:
        wait until (∀k ≠ i, Q[k] < j) or
   TURN[j] ≠ i
}
Critical section
Q[i] = 0
```

**Exhibit :** Simple n-Process Peterson's Mutual Exclusion Algorithm

It is based on repeated use of his two process algorithm. It requires 2n-1 shared variables of size n, etc. We have yet to complete a successful SPIN run using this form.

**References**


*Articles*

[1] 1981 – G. L. Peterson, "**Myths about Mutual Exclusion Problem**", Information Processing Letters, Vol. 12, number 3, pp. 115-116

[2] 2003 – Vikram Goenka, "**Peterson's Algorithm in a Multi Agent Database System**", Research Paper for 433-481, Knowledge Representation and Reasoning, July

*Books*

[9] **Modern Operating Systems, second edition**
Andrew S. Tanenbaum, Prentice Hall International, 2001, ISBN 0-13-092641-8

*Web based materials*

[15] **A Simpler Solution: Peterson's Algorithm**
*Jim Mooney, Ph. D. The Ohio State University*
http://www.csee.wvu.edu/~jdm/classes/cs356/notes/mutex/Peterson.html

[22] **Operating Systems: Steps towards Peterson's Mutual Exclusion Algorithm**
*Dr. Robert Kline, West Chester University*
http://www.cs.wcupa.edu/~rkline/OS/Peterson.html

[23] **Proof of a Solution to the Critical Section Problem**
*Jayson Rock*
http://www.cs.uwm.edu/classes/cs537/CS_AlgProof.html

[24] **Typical critical section program**
*Gonzalo Ramos, Ph.D. candidate at the University of Toronto*
http://www.dgp.toronto.edu/~bonzo/ta209/peterson.txt

[25] **Wikipedia, the Free Encyclopedia**
http://en.wikipedia.org