Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Processing Letters 96 (2005) 36–40

Information Processing Letters

www.elsevier.com/locate/ipl

A mutual exclusion algorithm with optimally bounded bypasses

K. Alagarsamy

Department of Computer Science, University of Northern British Columbia, Prince George, BC, Canada V2N 4Z9

Received 4 August 2004

Available online 11 July 2005

Communicated by F. Dehne

Abstract

Peterson's algorithm [G.L. Peterson, Myths about the mutual exclusion problem, Inform. Process. Lett. 12 (3) (1981) 115–116] for mutual exclusion has been widely studied for its elegance and simplicity. In Peterson's algorithm, each process has to cross $n-1$ stages to access the shared resource irrespective of the contention for the shared resource at that time, and allows unbounded bypasses. In [K. Block, T.-K. Woo, A more efficient generalization of Peterson's mutual exclusion algorithm, Inform. Process. Lett. 35 (1990) 219–222], Block and Woo proposed a modified algorithm that transforms the number stages to be crossed from fixed $n-1$ to t, where $1 \leqslant t \leqslant n$, and bounds the number of possible bypasses by $n(n-1)/2$. This paper proposes a simple modification that reduces the bound on the number of possible bypasses to optimal $n-1$.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Mutual exclusion; Critical section; Read/write atomicity; Bypasses; Synchronization; Concurrency

1. Background

The mutual exclusion problem is one of the important problems in concurrent programming. Dekker was the first to give a correct software solution to mutual exclusion problem for two processes case. Subsequently, Dijkstra proposed a solution for n processes [4]. Following Dijkstra's solution, many solutions appeared in the literature for n processes [11,2]. Ever since Peterson published his solution in 1981 [10], it has become de facto example of mutual exclusion algorithm for many researchers and academicians due to its simplicity and elegance. Also, the algorithm has been extensively studied and expanded in the literature [5,11,6,3,8,13,12,1,7,9,14]. In this paper, we first study Peterson's algorithm [10] and the algorithm by Block [3]. Then we propose another algorithm, which optimally bounds the number of possible bypasses.

The number of bypasses over a process indicates how long that a process, after showing its interest to access the shared resource, has to wait. Peterson's algorithm does not assure any bound on the number of possible bypasses over a process, in accessing the shared resource. Block–Woo's algorithm has the bound $n(n-1)/2$, which is still high. The algorithm

E-mail address: csalex@unbc.ca (K. Alagarsamy).

0020-0190/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2005.05.015

presented in this paper reduces the bound to $n - 1$, which is optimal for the worst case in this context.

The rest of the paper is organized as follows. The system model and the problem statement are given in Section 2. Section 3 analyzes Peterson's algorithm and the algorithm by Block and Woo. A new algorithm for mutual exclusion assuring optimal bound on the number of possible bypasses is presented in Section 4. Section 5 concludes the paper.

2. System model and problem statement

We assume a system of n independent cyclic processes competing for a shared resource R. In a process p, the part of the code segment that accesses R is called *a critical section* (*CS*) *of p for the resource* R [4]. The *mutual exclusion problem* is to design an algorithm that assures the following properties:

- *Safety*: At any time, at most one process is allowed to access the shared resource. Equivalently, at any time, at most one process is allowed to be in the CS.
- *Liveness*: When one or more processes have expressed their intentions to enter the CS, one of them eventually enters.

In addition, it is desirable to have the following property.

- *Freedom from Starvation*: Any process that expresses its intention to enter the CS will be able to do so in finite time.

The following assumptions are made in most mutual exclusion algorithms in this context.

Assumption 2.1. The execution speed of any process is finite but unpredictable.

Assumption 2.2. The solution must be symmetrical between all the processes; as a result we are not allowed to introduce a static priority.

Assumption 2.3. Each process begins its execution and subsequently halts, after possibly many cycles, only in its non-critical section.

3. Algorithms review

We begin with Peterson's algorithm. The basic idea behind Peterson's algorithm, given in [10], is that each process has to pass through $n - 1$ stages before entering the CS. These stages are designed to block one process per stage so that after $n - 1$ stages only one process will be eligible to enter the CS (which we consider as stage n).

The algorithm uses two integer arrays *TURN* and Q of sizes $n-1$ and n, respectively, and two integer local variables j and k. The value at $TURN[j]$ indicates the *latest* process entered at stage j, and $Q[i]$ indicates the latest stage that the process i is crossing through. The array Q is initialized to 0. The process id's, *pid*s, are assumed to be integers between 1 and n. The algorithm uses two conditions to cross a stage: $(TURN[j] \neq i)$ and $(\forall k \neq j,\ Q[k] < j)$—we call them, respectively, as *primary* and *secondary* conditions for crossing the stage j, for our reference.

A limitation of Peterson's algorithm is that each process must cross $n - 1$ stages, even if it is the only process competing for the CS. In [3], Block and Woo presented an interesting variation in which the number of stages to be crossed varies from 1 to n rather than fixed $n - 1$.

The idea behind Block–Woo's algorithm is that a process can enter the CS if it reaches a stage "high enough" rather than n always. For example, if there are m processes interested in CS accesses, then the process that reaches the stage m can enter the CS.

Definition 3.1. A stage j is said to be *high enough* to enter the CS if the total number of competing processes is less than or equal to j.

Since each competing process in Block–Woo's algorithm needs to compute only the number competing processes, to check whether it is in a stage high enough to enter the CS, the algorithm replaces the integer array Q of Peterson's algorithm with binary array *in* of same size. (If $in[i]$ is 1 then the process i is competing, otherwise i is non-competing.) Also, the algorithm increases the size of *TURN* from $n-1$ to n, and renamed it as *block*. For uniformity, we use the arrays' names as Q and *TURN* in this paper. There is no change in the primary condition in crossing the stage. The sec-

```
Process i:
1. for $j = 1$ to $n - 1$ do
2.   $Q[i] := j$;
3.   $TURN[j] := i$;
4.   wait until $(TURN[j] \neq i) \vee (\forall k \neq i, Q[k] < j)$
5. enddo;
6. $CS$;
7. $Q[i] := 0$;
```

Fig. 1. Peterson's Algorithm [10].

```
Process i:
  $j := 0$;
1. $Q[i] := 1$;
2. repeat
     $j := j + 1$
3.   $TURN[j] := i$;
4.   wait until $(TURN[j] \neq i) \vee (\sum_{k=1}^{n} Q[k] \leqslant j)$
5. until $(TURN[j] = i)$
6. $CS$;
7. $Q[i] := 0$;
```

Fig. 2. Block–Woo's Algorithm [3].

ondary condition to cross the stage j of Block–Woo's algorithm is $(\sum_{k=1}^{n} Q[k] \leqslant j)$.

The code segments for process i of Peterson's algorithm and Block–Woo's algorithms are given in Figs. 1 and 2.

3.1. Analysis

The main differences between Peterson's algorithm and Block–Woo's algorithm are that (i) the fixed iterative control structure (for-loop) of Peterson's algorithm is replaced in Block–Woo's algorithm by the nondeterministic control structure (repeat-until) and (ii) the explicit position check using exact Q values of other processes in the secondary condition of Peterson's algorithm is replaced in Block–Woo's algorithm by an implicit position check in terms of counting the non-zero Q values. With these changes, Block–Woo's algorithm changes the number of stages to be crossed from fixed $n - 1$ to variable t $(1 \leqslant t \leqslant n)$ and bounds the possible bypasses by $n(n - 1)/2$.

Consider the following scenario. All n processes start their competition for the CS almost simultaneously and one process, say p_1, gets blocked at stage 1 (here $n - 1$ bypasses over p_1 are possible. This is because, among all n competing processes, the process p_1 might complete its first write operation to show its interest in CS access but eventually blocked at stage 1 by setting *TURN* last) and all other processes are bumped to the upper stages; the process that reaches the stage n enters the CS; after that process leaves the CS the other $n - 2$ processes enter and leave the CS, one by one (during this period p_1 is blocked at stage 1); after all $n - 1$ processes complete their CS accesses and just before p_1 can make a move, the $n - 1$ process come back quickly and enter stage 1 (now $n - 2$ processes may bypass p_1 leaving one process to be blocked at stage 1, and p_1 could be blocked at stage 2); and this pattern may continue, each time p_1 can move one stage up with one less number of bypasses. Summing up the number of possible bypasses, we get $n(n - 1)/2$.

The huge bypasses occur because p_1 is blocked at stage 1 without any apparent promotion in its position to access the CS, for the entire duration in which all the processes in the upper stages complete their CS accesses.

4. Algorithm with optimal bypasses

Our main objective in this paper is to refine Peterson's algorithm to have the property of assuring optimal bound on the number of bypasses. The basic idea behind our algorithm is that every process, after completing its CS access, should *promote* the other processes waiting for the CS. This is done by releasing the waiting processes between the stages 1 and $j - 1$ inclusive, where j is the stage from which the current promoter entered the CS. It is apparent that the probability and the number of bypasses decrease as a process moves up to the higher stages.

Promoting the processes along the stages might cause more than one process to reach the stage high enough to enter the CS. To handle this problem, the secondary condition of our algorithm combines the secondary conditions of both Peterson's and Block–Woo's algorithms. That is, in our algorithm, *a process can enter the CS if it reaches a stage, say j, high enough to enter the CS* and *all other processes are in the stages below j*.

We use the same set of variables used in Peterson's algorithm, with only change that the size of *TURN*

```
Process i:
   $j := 0$;
 1. repeat
    $j = j + 1$;
 2. $Q[i] := j$;
 3. $TURN[j] := i$;
 4. wait until $(TURN[j] \neq i) \vee ((\forall k \neq i, Q[k] < j)$
                  $\wedge (|\{Q[k]/\forall k, Q[k] \neq 0\}| \leqslant j))$
 5. until $(TURN[j] = i)$
 6. $CS$;
 7. for $k = 1$ to $j - 1$ do
 8.    $TURN[j] := i$;
 9. wait until $(\forall k \neq i, (Q[k] = 0) \vee (TURN[Q[k]] = k))$
10. $Q[i] := 0$;
```

Fig. 3. The algorithm with optimal bypasses.

is increased from $n - 1$ to n. There is no change in the primary condition to cross a stage. The secondary condition of our algorithm is $(\forall k \neq i, Q[k] < j) \wedge (|\{Q[k]/\forall k, Q[k] \neq 0\}| \leqslant j)$. The code segment of our algorithm for process i is given in Fig. 3.

The **wait until** statement at line 9 is to ensure that the promotion is achieved.

4.1. Correctness

We first introduce some definitions.

Definition 4.1. A process i *starts competing for the CS* when it sets $Q[i]$ to 1 at line 2.

Definition 4.2. A process is *at stage* j after setting $TURN[j]$ at line 3.

Definition 4.3. A process is *blocked at stage* j until it has completed its execution of line 4.

Definition 4.4. A process i setting $TURN[j] = i$ at line 3 is called *bumping the process* at stage j.

Definition 4.5. A process i setting $TURN[j] = i$ at line 8 is called *releasing the process* at stage j for *promotion.*

Definition 4.6. A process i completed the execution of the statement at line 4 with $Q[i]$ value j is said to have *crossed* the stage j.

Observation 4.1. *A process cannot reach the stage* j $(j > 1)$ *unless it is either bumped or released for promotion from stage* $j - 1$.

A process i which crossed the stage $j - 1$ without either bumped or released at stage j will exit from the repeat-until loop, because $TURN[j - 1] = i$ when executing the line 5.

Lemma 4.1. *No process will reach a stage beyond* n.

Proof. No process can reach the stage beyond n by bumping alone, because for a process to get to the stage beyond n by bumping must involve at least $n + 1$ processes. Since no promotion is possible before the first entry to the CS, the first entry to the CS can occur only from the stage n or below. Assume that p is in the CS after entered from the stage j. The promotion of waiting processes by p will not cause any process go beyond the stage j. This is because p promotes only the processes in the stages between 1 and $j - 1$ inclusive, one stage up. In general, at any time, no promotion can cause a process to move beyond the stage of the promoter. By the inductive argument on the index of CS entries, no process will reach a stage beyond n. □

Lemma 4.1 assures the required sizes of *TURN* and Q as n.

Theorem 4.1. *The algorithm given in Fig.* 3 *assures mutual exclusion.*

Proof. Proof by contradiction. Assume that the process p and q are in the CS and p entered first from stage i, q entered later from stage j. Clearly $i < j$, because $Q[q]$ should be greater than $Q[p]$ for q to exit from the statements at line 4 and then subsequently at line 5. Since no release operation is possible before p completes its CS access, by Observation 4.1, all the stages between 1 and $j-1$ need to be occupied by the competing processes to bump q to the stage j. When p made its decision to enter the CS from the stage i, no other processes might have been in stage i or above. That means all the stages between 1 and $j - 1$ must be occupied by the processes other than p and q, when q entered the CS from the stage j. This implies that, adding with the processes p and q, totally there must be at least $j+1$ processes competing for the CS when q entered the CS from the stage j. This is impossible, because q was not in a stage high enough (that is, the condition $|\{Q[k]/\forall k, Q[k] \neq 0\}| \leqslant j$ is not true) to enter the CS. This completes the proof. □

Theorem 4.2. *The algorithm given in Fig.* 3 *is free from deadlock.*

Proof. Assume that m processes are competing for the CS. Since the primary condition $(TURN[j] \neq i)$ can block at most one process per stage, the competing processes have to eventually spread to occupy m distinct stages. Then at least one process will be in a stage, say k, high enough to enter the CS ($k \geqslant m$), and all other processes will be in stages below j. This process, say p, will enter the CS eventually. This is because, for p, the secondary condition is true (so it can exit from **wait until** statement at line 4) and primary condition is false (therefore it can come out of **repeat-until** loop). □

Theorem 4.3. *The algorithm given in Fig.* 3 *assures* $n-1$ *as the bound on the number of possible bypasses over a process.*

Proof. Assume the worst possible situation that all the n processes start competing for the CS almost simultaneously and the process, say p, which started first among all is blocked at stage 1. Here after, each process that bypassed p at stage 1 assures p to move one stage up by promoting it after completing its CS access. Even if no newly competing process bumped p to go up, after $n-1$ CS accesses the process p will reach the stage n and therefore it can enter the CS. Hence the proof. □

Corollary 4.1. *The algorithm given in Fig.* 3 *is free from starvation.*

Proof. Follows from Theorem 4.3 □

5. Concluding remarks

We have presented a mutual exclusion algorithm that assures optimal bound on the number of possible bypasses. Peterson's algorithm and Block–Woo's algorithm are relatively simple compared to our algorithm. But our algorithm assures improved fairness by optimally bounding the number of bypasses. Our algorithm acquired its complexity in an effort to achieve optimal bypasses. In fact, we obtained our algorithm as a result of an effort to derive the optimal bound on the number of possible bypasses for Block–Woo's class of algorithms.

References

[1] K. Alagarsamy, K. Vidyasankar, Fair and efficient mutual exclusion algorithms, in: Lecture Notes in Computer Science, vol. 1693, Springer, Berlin, 1999, pp. 166–179.
[2] J.H. Anderson, Y.J. Kim, T. Herman, Shared-memory mutual exclusion: major research trends since 1986, Distrib. Comput. 16 (2003) 75–110.
[3] K. Block, T.-K. Woo, A more efficient generalization of Peterson's mutual exclusion algorithm, Inform. Process. Lett. 35 (1990) 219–222.
[4] E.W. Dijkstra, Solution of a problem in concurrent programming control, Comm. ACM 8 (9) (1965) 569.
[5] E.W. Dijkstra, An assertional proof of a program by G.L. Peterson, in: EWD 779, 1981.
[6] M. Hofri, Proof of a mutual exclusion algorithm—A classic example, ACM SIGOSR 24 (1) (1990) 18–22.
[7] Y. Igarashi, Y. Nishitani, Speedup of the n-process mutual exclusion algorithm, Parallel Process. Lett. 9 (4) (1999) 475–485.
[8] J. Misra, A family of 2-process mutual exclusion algorithms, in: Notes on Unity, The University of Texas at Austin, 1994.
[9] K. Obokata, M. Omari, K. Motegi, Y. Igarashi, A lockout avoidance algorithm without using time-stamps for k-exclusion problem, in: Lecture Notes in Computer Science, vol. 2108, Springer, Berlin, 2001, pp. 571–575.
[10] G.L. Peterson, Myths about the mutual exclusion problem, Inform. Process. Lett. 12 (3) (1981) 115–116.
[11] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, Cambridge, MA, 1986.
[12] Y. Tsay, Deriving scalable algorithms for mutual exclusion, in: Symp. on Distributed Computing, 1998, pp. 393–407.
[13] F.W. van der Sommen, W.H.J. Feijen, A.J.M. van Gasterm, Peterson's mutual exclusion algorithm revisited, Sci. Comput. Programming 29 (1997) 327–334.
[14] K. Vidyasankar, A simple group l-exclusion algorithm, Inform. Process. Lett. 85 (2003) 79–85.