Some Myths About Famous Mutual Exclusion Algorithms*

K. Alagarsamy
Department of Computer Science
University of Northern British Columbia
Prince George, British Columbia, Canada
E-mail: *csalex@unbc.ca*

August 5, 2003

Abstract

Dekker's algorithm[9] is the historically first software solution to mutual exclusion problem for 2-process case. The first software solution for n-process case was subsequently proposed by Dijkstra[8]. These two algorithms have become de facto examples of mutual exclusion algorithms, for their historical importance. Since the publication of Dijkstra's algorithm, there have been many solutions proposed in the literature[24, 1, 2]. In that, Peterson's algorithm[21] is one among the very popular algorithms. Peterson's algorithm has been extensively analyzed for its elegance and compactness. This paper attempts to dispel the myths about some of the properties of these three remarkable algorithms, by a systematic analysis.

1 Introduction

1.1 Background

The mutual exclusion problem is well known for more than four decades since it was posed as early as 1962[8]. Dekker was the first to propose a correct software solution to this problem for 2-process case [9]. Subsequently, Dijkstra proposed the first solution for n-process setting. These two algorithms have become de facto examples of mutual exclusion algorithms for many researchers and most text books on operating systems and concurrent programming, due to their historical importance. Later, there have been many solutions proposed in the literature for mutual exclusion problem[24, 1, 2]. Among the published solutions in the literature, Peterson's algorithm[21] is a popular one. Peterson's algorithm has been extensively analyzed for its elegance and compactness.

*Copyright©K.Alagarsamy, 2003

1.2 Motivation

Concurrency has become an inevitable phenomenon of present-day computing. The technological developments in the software and hardware industries and users' demands in various fields paved the way to incorporate concurrent and distributed processing in most systems and applications. Therefore, teaching concurrent programming is becoming an integral part of most computer science programs.

Concurrent programs are extremely hard to design[1] and notorious for common errors. In this context, any misleading interpretations or references about popular solutions will only add further complexity to the subject matter.

1.3 Contributions of this Paper

The mutual exclusion algorithms by Dekker[9], Dijkstra[8, 9], and Peterson[21] are extensively analyzed and referred in most text books and research report's for their merits. Yet a few properties of these algorithms have eluded the researchers and academicians. This paper systematically analyzes the algorithms presented in [8, 9, 19, 21] and dispels the myths about some of their properties. Also, a generalization of Dekker's algorithm is presented.

1.4 Organization

The rest of the paper is organized as follows. Section 2 presents the system model and problem statement. Dekker's algorithm is reviewed in section 3. The concept of generalization and referred generalizations of Dekker's algorithm are analyzed in Sections 4 and 5. Section 6 presents a generalization to Dekker's algorithm, and the bypass property of Peterson's algorithm is examined in Section 7. The paper is concluded in Section 8.

2 System Model and Problem Statement

We assume a **system** of n independent cyclic processes competing for a shared resource. The code segment of a competing process can be divided into two parts: the part which accesses the shared resource (*Critical Section (CS)*) and the remaining part (*Noncritical Section (NCS)*). **The mutual exclusion problem** is to design an algorithm that assures the following properties: At any time, at most one process is allowed to be in CS (*Safety*); When one or more processes have expressed their intentions to enter CS, one of them eventually enters (*Liveness*). In addition to these two properties the following is a desirable property: Any process that expresses its intention to enter CS will be able to do so in finite time (*Freedom from Starvation*).

[1]The problem has been solved for two processes by T.J. Dekker in the early sixties. It has been solved by me for the N processes in 1965. The solution for two processes was complicated, the solution for N processes was terribly complicated. (The program pieces for "enter" and "exit" are quite small, but they are by far the most difficult pieces of program I ever made). - Dijkstra[11]

3 Dekker's Algorithm

The original Dekker's algorithm uses many "goto" statements which are "considered harmful" [10], and almost vanished from present-day programming. Therefore, we will rewrite Dekker's algorithm using modern control structures.

The basic idea behind the implementation of Dekker's algorithm is that a process enters CS straight away when the other process is not competing for CS. When both processes are simultaneously interested in accessing CS, the tie is broken by allowing the process which accessed CS least recently to succeed. The algorithm uses three binary variables, $c1$, $c2$, and *turn*. For a better readability we rename the first two variables, respectively, as $state[1]$ and $state[2]$. Also, we define *other* as $3 - i$ where i may take value either 1 or 2, *out* as 0, and *competing* as 1. The *status* bits are initialized to *out*. The formal code is presented in Figure 1.

```
1.    status[i] := competing;
2.    while(status[other] = competing)
3.    {
4.       if(turn = other)
5.       {
6.          status[i] := out;
7.          wait until(turn = i);
8.          status[i] := competing;
9.       }
10.   }
11.   CS;
12.   turn := other;
13.   status[i] := out;
```

Figure 1: Dekker's algorithm (*DKA*)

Theorem 3.1 *Dekker's algorithm is free from starvation.*

Proof: It is easy to see that a process enters CS straight away when the other process is not competing for CS. Assume that both processes 1 and 2 are simultaneously interested in CS, and *turn* = 1. Then both will enter the **while** loop but only process 2 will enter the **if** block at line 4. Subsequently, 2 will set its *status* to *out*, and wait for its *turn*, at line 7. Now 1 can exit from the **while** loop and proceed further to access CS. After completing CS execution the process 1 will set the *turn* to 2. Then process 2 will eventually observe the *turn* value as 2 and set its status to *competing*. At this stage, even if the process 1 comes back again it will be blocked at line 7 yielding the way for 2 to go ahead and access CS. Hence the proof.

4 The Concept of Generalization

Generalization is a powerful tool used to devise solutions to complex problems since time immemorial. The idea is that first deal with something familiar and concrete version of the problem that is easy to work with. If a solution is obtained for the simplified version then with that experience the abstract properties of the problem (that is, the more generic cases) may be treated easily. The advantage with this approach is that in many cases the observations from the solution of the concrete or simplified case will lead to easily extend the obtained solution to solve the more general case. Unfortunately, in many cases such a generalization of solutions is difficult or impossible and requires different approach to obtain the solution for the general case of the problem. We reproduce two popular definitions of generalization, below, for our reference.

Definition 4.1 *Generalization is passing from the consideration of a restricted set to that of a more comprehensive set* **containing** *the restricted one. - George Polya[23].*

Definition 4.2 *A method of generalization is not uniquely determined, for there are usually numerous ways of carrying it out. One requirement, however, must be rigorously satisfied: any generalized concept must* **reduce to the original one** *when the original conditions are fulfilled.*

- Albert Einstein and Leopold Infeld [12].

From the above two definitions of generalization, we can easily infer that any generalization should logically *include the original one.*

5 Referred Generalizations of Dekker's Algorithm - An Analysis

Dijkstra has never claimed his algorithms [8, 9] as generalizations of Dekker's algorithm. However, many researchers and text books on concurrency and operating systems [3, 4, 1, 6, 5, 7, 14, 16, 17, 19, 20, 18, 22, 24, 25, 26, 2] refer the algorithms presented in [8, 9] as generalizations or extensions of Dekker's algorithm. In [19] Martin presented an algorithm claiming it as another generalization of Dekker's algorithm. In this section we will show that the referred generalizations of Dekker's algorithm do not satisfy the basic requirement of definition of generalization, discussed in Section 4. In the algorithm presented in [8] (we call as $DJA1$) the *turn* is not initialized, and left unchanged after each CS execution. In the algorithms presented in [9, 19] (we call as $DJA2$ and MRA) the *turn* is initialized to 0, and set to 0 after each CS access. We present only $DJA1$ and MRA because the change of $DJA2$ with $DJA1$ is very minimal, that *turn* := 0 is added at the end in $DJA2$.

Theorem 5.1 *The algorithm $DJA1$ is susceptible to starvation.*

```
1.  $status[i] := competing$;
2.  **do**
3.  {
4.      **while**($turn \neq i$)
5.      {
6.          $status[i] := out$;
7.          **if** $status[turn] = out$ **then** $turn := i$;
8.      }
9.      $status[i] := cs$;
10. }**while**($status[other] = cs$);
11. **critical section;**
12. $status[i] := out$;
```

Dijkstra's Algorithm ($DJA1$)

```
1.  $status[i] = competing$;
2.  **while**($status[other] = competing$)
3.  {
4.      $status[i] := out$;
5.      **wait until**($turn = 0 \vee turn = i$);
6.      $turn = i$;
7.      $status[i] = competing$;
8.  }
9.  **critical section;**
10. $status[i] := out; turn := 0$;
```

Martin's Algorithm (MRA)

Proof: In $DJA1$ the tie is broken by favoring "most recently" CS accessed process (that is, $turn$ will be always with the process which accessed CS most recently). This policy obviously leads to starvation if the process, which holds the current $turn$, is continuously interested in accessing CS.

Theorem 5.2 *The algorithm MRA is susceptible to starvation.*

Proof: Consider the situation where both processes 1 and 2 are simultaneously interested in CS. Assume that 1 succeeds in the tie breaking contest by successfully capturing the $turn$, and goes ahead to access CS. Suppose the process 2 waits at line 5, for $turn$ to become "0", at times $t_1, t_2, t_3, ...$ In the mean time process 1 can repeatedly access CS by setting the $turn$ to 0 and 1, alternatively, in such a way that $turn$ value is "1" at times $t_1, t_2, t_3, ...$ Since the processes speeds are arbitrary this scenario may be quite possible. Therefore, i may never come out of the **while** loop at line 2. Hence the proof.

Theorem 5.3 *The algorithm $DJA2$ is susceptible to starvation.*

Proof is similar to the proof of Theorem 5.2.
One of the very important invariant properties of Dekker's algorithm is that the algorithm is free from starvation. But $DJA1, DJA2$, and MRA do not assure this property, even when $n = 2$. Also, during tie breaking in Dekker's algorithm the process which accessed CS least recently always wins. In $DJA1$ the "most recently" CS accessed wins the competition, and in $DJA2$ and MRA the tie breaking is nondeterministic. The key behavioral differences among the four algorithms are summarized in Table 1.

The differences between Dekker's algorithm and the other three algorithms are significant even when $n = 2$, and that violate the basic requirement of generalization which we discussed in Section 4. Hence the algorithms $DJA1, DJA2$, and MRA cannot be considered as generalizations of Dekker's algorithm.

Myth 1 *Dijkstra's algorithms* [8, 9] *and Martin's algorithm* [19] *are generalizations of Dekker's algorithm.*

	Properties	*DKA*	*DJA1*	*DJA2*	*MRA*
1	Is susceptible to starvation, **when** $n = 2$ **?**	No	Yes	Yes	Yes
2	Is tie breaking deterministic ?	Yes	Yes	No	No
3	Initial value of *turn*	any *id*	any value	0	0
4	While leaving CS *turn* is set to	other	(Left as) its own	0	0

Table 1: A Comparison of Dekker's Algorithm and Its Referred Generalizations

6 A Generalization of Dekker's Algorithm

In this section, we present a generalization to Dekker's algorithm called *GDKA*. Our generalization uses a boolean array $status[n]$ and an integer array $turn\_pos[n]$. The *turn_pos* array holds relative order of the past CS accesses. We present two functions in Figure 3: $AP\_Empty(k)$ - to check whether there is any other process with higher priority currently competing for CS and; $Adjust\_LRA(k)$ - to adjust the priorities of the processes after completing CS.

The core part of the algorithm is how the tie is broken when more than one process are simultaneously competing for CS. The process which accessed CS "least recently" among the competing processes is selected to go first. The formal code of the algorithm is presented in Figure 3.

```
GDKA(i):
0.   $k := n$;
1.   $status[i] := competing$;
2.   while($\exists j \neq i, status[j] = competing$);
3.   {
4a.    while($turn\_pos[k] \neq i \wedge k > 1$) $k := k - 1$;
4b.    if ($AP\_Empty[k] = yes$)
5.     {
6.       $status[i] := out$;
7.       wait until($turn\_pos[k] \neq i$)
8.       $status[i] := competing$;
9.     }
10.  }
11.  CS;
12.  $Adjust\_LRA(k)$;
13.  $status[i] := out$;
```

```
AP_Empty(k):
1.   $found := no$;
2.   for $j := 1$ to $k - 1$ do
3.     if($status[turn\_pos[j]] \neq out$)
         $found := yes$;
4.   $return(found)$
```

```
Adjust_LRA(k):
1.   for $j := k$ to $n - 1$ do
2.       $turn\_pos[j] := turn\_pos[j+1]$;
3.   $turn\_pos[n] := i$;
```

Figure 3.

The correctness proofs of the algorithm is beyond the scope of this paper. In this paper, we only justify that *GDKA* is a generalization of Dekker's algorithm. For that, we need to show that *GDKA* is behaviorally equivalent to Dekker's algorithm when $n = 2$. We will compare the Dekker's algorithm (DKA) with our generalization $GDKA$, line by line for the case $n = 2$.

Let the ids of two competing processes be i and j: Line 1 of $GDKA$ is same as the line 1 of DKA; Line 2 of $GDKA$ is equivalent to line 2 of DKA, because the "other" process different from i is only j; Lines 4a and 4b of $GDKA$ are mainly to check whether any other processes with higher priority are currently competing for CS. In 2-process case it is obvious that the other process is competing (otherwise this process wouldn't have entered into the **while** loop at line 2 of $GDKA$). Therefore we need only to check for *turn* value. This is what exactly the line 4 of DKA does. So, the lines 0, 4a and 4b of $GDKA$ are collectively equivalent to the line 4 of DKA when $n = 2$; Line 6 of $GDKA$ is same as line 6 of DKA; A process might have entered into the **if** block at line 4b of $GDKA$ (line 4 of DKA) only if the *turn* is set to the other process. After a CS execution the *turn* value must change. The line 7 of both the algorithm is designed to force a process to wait for a *turn* change. Therefore the line 7 of $GDKA$ and DKA is semantically same; Line 8 of $GDKA$ is same as the line 8 of DKA; The function *Adjust_LRA* puts the current process as the most recently CS accessed and the other process as least recently CS accessed. This is exactly equivalent to setting the *turn* to the other process. Therefore the function *Adjust_LRA*() of $GDKA$ is equivalent to the line 12 of DKA; Line 13 of $GDKA$ is same as the line 13 of DKA. Therefore, $GDKA$ may be considered as a generalization of Dekker's algorithm. Note that our generalization requires significant effort and structural changes and we are not aware of any other generalizations in the literature, other than the inaccurately referred as generalizations of Dekker's algorithm[8, 9, 19].

Myth 2 *Dekker's algorithm can be trivially modified to solve n-process case.*

7 Peterson's Algorithm and Unbounded Bypass

Unbounded bypass of the n-process tournament algorithm, whose components are Peterson's algorithm for the case $n = 2$, is widely known [17]. But there is a confusion about this property in Peterson's algorithm presented in [21] that it assures bounded bypass [24, 15, 13]. But apparently Peterson's algorithm does not assure bounded bypass, if $n > 2$.

The basic idea behind Peterson's n-process mutual exclusion algorithm [P81] is that each process passes through $n-1$ stages before entering critical section. These stages are designed to block one process per stage so that after $n - 1$ stages only one process will be eligible to enter critical section (which we consider as stage n). The algorithm uses two integer arrays *step* and *pos* of sizes $n - 1$ and n respectively. The value at *step*$[j]$ indicates the *latest* process at step j, and *pos*$[i]$ indicates the latest stage that the process i is passing through. (Peterson uses Q for *pos*, and $TURN$ for *step*.) The array *pos* is initialized to 0. The code segment for process i is given in Figure 4.

Next we explain through following scenario that Peterson's algorithm does not assure bounded bypass.

The processes p_1, p_2, and p_3 are currently competing for CS: Process p_1 starts first, sets *pos*$[p_1] = 1$; p_3 starts and assigns its *pid* to *step*$[1]$; p_2 sets its *pid* to *step*$[1]$ and so p_3 is

```
Process i:
1. for $j = 1$ to $n - 1$ do
2. {
3.   $pos[i] := j$;
4.   $step[j] := i$;
5.   wait until $(\forall k \neq i, pos[k] < j) \vee (step[j] \neq i)$
6. }
7. cs.i;
8. $pos[i] := 0$;
```

Figure 4 : Peterson's Algorithm

pushed; since the condition $(\forall k \neq p_2, pos[k] < 1) \vee (step[1] \neq p_2)$ is not true, p_2 is blocked at stage 1; p_3 crosses stage 1 to stage 2; since the condition $(\forall k \neq p_3, pos[k] < j)$ is true, for $j \geq 2$, it keeps proceeding further, enters CS, and completes CS execution; p_3 starts competing again for CS, and sets its *pid* to *step*[1]; the condition $(step[1] \neq p_2)$ becomes true, that is, p_2 is unblocked and p_3 is blocked at stage 1; p_2 moves up all the way, enters and leaves CS, starts competing again, and sets its *pid* to *step*[1]; this time p_2 gets blocked and p_3 is unblocked, at stage 1 and; p_2 and p_3 can overtake p_1, alternately, several times until p_1 sets $step[1] = p1$. This implies that unbounded bypass is possible in Peterson's algorithm.

Myth 3 *Peterson's algorithm [21] assures bounded bypass.*

8 Conclusion

Dijkstra has never claimed his algorithms [8, 9] as generalizations of Dekker's algorithm. However, many researchers and text books on concurrency and operating systems [3, 4, 1, 6, 5, 7, 14, 16, 17, 19, 20, 18, 22, 24, 25, 26, 2] refer the algorithms presented in [8, 9] as generalizations or extensions of Dekker's algorithm. These references might have led Martin to call his algorithm [19] as another generalization of Dekker's algorithm. Even the recent references [1, 2, 20] confirm such a myth that Dijkstra's algorithm is a generalization of Dekker's algorithm.

Also, Peterson has never claimed that his algorithm assures bounded bypass. But Raynal [24] calculated the maximum possible bypass in Peterson's algorithm as $n(n-1)/2$. Kowaltowski and Palma [15] believed it to be $n - 1$ and Hofri [13] derived it to be $n - 1$ under certain liveness assumptions. Problem 10.12 in Lynch's book [17], page 329, asks for checking whether bounded bypass is guaranteed. From the results in [15, 24, 13] one might incorrectly infer that bounded bypass is guaranteed in Peterson's algorithm.

Myths 1 and 2 are not newly observed. They are mentioned in [21] without elaboration. However, these observations have been either ignored or incorrectly reflected in the later part of the literature and text books on concurrency and operating systems.

Concurrent programs are notorious for subtle errors, and slips are often possible while characterizing or proving the properties of a concurrent program. We feel that these types of errors or inaccuracies should be disseminated to the scientific community as soon as they are identified. And that will increase the clarity of the subject matter and help to eliminate cascading or perpetual errors in concurrent programs.

References

[1] J. Anderson, "Lamport on Mutual Exclusion: 27 Years of Planting Seeds", PODC, 3-12, 2001.

[2] J. Anderson and Y. J. Kim, Shared-memory Mutual Exclusion: Major Research Trends Since 1986, Distributed Computing, To appear.

[3] G.R. Andrews, Concurrent Programming : Principles and Practice, The Benjamin / Cummings Publishing Company, 1991.

[4] G.R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, 2000.

[5] J. Bacon, Concurrent Systems, Addison-Wesley, 1999.

[6] M. Ben-Ari Principles of Concurrent Programming, PHI, 1982.

[7] L. Bic and A.C. Shaw, The Logical Design of Operating Systems, PHE, 1988.

[8] E.W. Dijkstra, Solution of a Problem in Concurrent Programming Control, CACM, Vol.8(9), Sept., 1965, 569.

[9] E.W. Dijkstra, Cooperating Sequential Processes (Techniche Hogeschool, Eindhoven, 1965). Reprinted in: F. Genuys (ed.), Programming Languages, Academic Press, 1968, 43-112.

[10] E.W. Dijkstra, Go To Statement Considered Harmful, CACM 11(3), 147-148, 1968.

[11] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. Acta Informatica, 1(2):115-138, 1971.

[12] Albert Einstein and Leopold Infeld, The Evolution of Physics - The Growth of Ideas from Early Concepts to Relativity and Quanta, Cambridge University Press, 1971.

[13] M. Hofri, Proof of a Mutual Exclusion Algorithm - A 'class'ic example, ACM SIGOPS OSR 24(1):18-22, 1990.

[14] R.C. Holt, E.D. Lazowska, G.S. Graham and M.A. Scott, Structured Programming with Operating Systems Applications, Addison-Wesley, 1972.

[15] T. Kowalttowski and A. Palma, Another Solution of the Mutual Exclusion Problem, IPL (19), 3, 145-146, 1984.

[16] S. Krakowiak, Principles of Operating Systems, The MIT Press, 1988.

[17] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers inc., 1996.

[18] M.Maekawa, E. Oldehoeft and R. Oldehoeft, Operating Systems: Adavanced Concepts, The Benjamin/Cummings Publishing Company, Inc., 1987.

[19] A.J. Martin, A New Generalization of Dekker's Algorithm for Mutual Exclusion, IPL 23(6), 1986, 295-297.

[20] A.J. Martin, Lecture Notes on "CS139: Concurrency in Computation", California Institute of Technology (www.async.caltech.edu/ cs139), 2003.

[21] G.L. Peterson, Myths About the Mutual Exclusion Problem, IPL 12(3) 1981, 115-116.

[22] J.R. Pinkert and L.L. Wear, Operating Systems Concepts, PHI, 1989.

[23] G. Polya, How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 1973.

[24] M. Raynal, Algorithms for Mutual Exclusion, MIT press, 1986.

[25] W. Stallings, Operating Systems: Internals and Design Principles, PHI, 1998.

[26] D.C. Tsichritzis and P.A. Bernstein, Operating Systems, Academic Press, 1976.