Intricacy of Concurrent Programming

K. Alagarsamy
Department of Computer Science
University of Northern British Columbia
Prince George, BC, Canada - V2N 4Z9
csalex@unbc.ca

Abstract

Concurrency has become an inevitable phenomenon of present-day computing. Therefore, teaching concurrent programming is increasingly becoming an integral part of most computer science programs. Concurrent programs are notorious for subtle errors. In this paper, we will attempt to explain the complexity involved in concurrent programming by examining some widely studied concurrent algorithms. We expose the inaccuracy or incorrectness in observing some of the basic properties of these algorithms through a careful analysis. We feel that this exercise would help the developers (instructors, students, and programmers) to understand the importance of the level of attention needed, while designing or studying concurrent programs.

Keywords: Concurrency, distributed coordination, and mutual exclusion.

1. Introduction

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 are paving the way to incorporate concurrent and distributed processing in most software systems and applications. Therefore, teaching concurrent programming is becoming an integral part of most computer science programs.

Mutual exclusion problem is one of the important concurrent programming problems investigated, as early as 1962[3]. In this paper, we analyze the mutual exclusion algorithms by Dekker[4], Dijkstra[3], Knuth[7], Lamport[9], and Peterson[12]. We choose these algorithms for our analysis because they are (1) thoroughly studied and extensively referred in most text books and research reports on this subject and (2) relatively simpler compared to most of the other later algorithms to solve mutual exclusion problem.

Concurrent programs are extremely hard to design[1] and notorious for subtle errors. We will attempt to explain the subtlety of concurrent programs by systematically exposing some of the evasive properties of these algorithms. The properties we will study are: *generalization*, *non-circularity,* and *bound on bypass*. First we will introduce the system and the problem statement followed by brief descriptions of the algorithms by Dekker, Dijkstra, Knuth, Lamport, and Peterson. Then we will analyze Dekker's algorithm and Dijkstra's algorithm for generalization concept, and the algorithms of Dekker, Knuth, and Lamport for non-circularity, and finally, Peterson's algorithm for bounded-bypass.

[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.

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:

1. 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 ***(Safety***).
2. When one or more processes have expressed their intentions to access the shared resource, one of them eventually accesses (***Liveness***).

In addition to these two ***essential properties***, the following are the ***desirable properties***:

3. Any process that expresses its intention to access the shared resource will be able to do so in *finite time* (***Freedom from Starvation***).
4. If any process expresses its intention to access the shared resource, then it will not be *bypassed* by other processes more than a *fixed number of times* to access the shared resource (***bounded bypass***).

The mutual exclusion algorithm has essentially two components: **entry code** and **exit code**. These pieces of codes have to be inserted before and after the CS into the code segment of each processes to ensure mutually exclusive access to the CS.

3. Algorithms Review

Here we briefly review the algorithms of Dekker, Dijkstra, Knuth, Lamport, and Peterson.

3.1 Dekker's Algorithm

In Dekker's algorithm, a process accesses the shared resource straight away when the other process is not competing for it. When both processes are simultaneously interested in accessing the shared resource the *tie* is broken by allowing the process which accessed the shared resource least recently to succeed.

The algorithm uses three binary variables c1, c2, and *turn*. The variables c1 and c2 are used to indicate the processes' status in the competition for the shared resource and the *turn* variable is used to break the tie. That is, the process which holds the *turn* is allowed to access the shared resource next. After completing the shared resource access the *turn* is given to the other process. For better readability, we rename c1 and c2 as *status[1]* and *status[2]*, define the variable *other* as *3-i* where *i* may take the value either 1 or 2, *out* as 0, and *competing* as 1. The *status* bits are initialized to out. The formal code for process *i* is given in Figure 1.

Process *i*:

```
status[i] := competing;
while(status[other] = competing)
{
        if(turn = other)
        {
                status[i] := out;
                wait until(turn = i)
                status[i] := competing;
        }
}
CS;
turn := other;
status[i] := out;
```

Figure 1. Dekker's Algorithm.

3.2 Dijkstra's Algorithm

The basic idea behind Dijkstra's algorithm is that: (i) a process which captures the turn successfully is allowed to access the shared resource; (ii) a process can capture the *turn* only if it is free. In a concurrent competition, more than one process may capture the *turn* simultaneously. Therefore, a process after capturing the *turn*, checks to see whether any other process has grabbed the *turn* after that. If so, then it restarts its competition for *turn* again. Since the *turn* variable can hold only one value at a time, eventually one process will succeed in capturing it and that process will advance further to access the shared resource. The 2-process version of Dijkstra's algorithm is given in Figure 2.

Process i:

```
status[i] := competing;
do
{
        while(turn ≠ i)
        {
                status[i] := out;
                if(status[turn] = out) then turn := i;
        }
        status[i] := cs;
} while(status[other] = cs);
CS;
status[i] := out;
```

Figure 2. Dijkstra's Algorithm (2-process version).

3.3 Knuth's Algorithm

In Knuth's algorithm, all the processes are kept in a *logical circle* and the *turn* goes around the circle giving each process a chance to access the shared resource. If the process corresponds to the current *turn* is not interested in accessing the shared resource, then the *turn* goes to the next closest competing process. We denote *HPP(i)* as the set of processes with priority higher than that of *i*. The algorithm is given in Figure 3.

Process i:

```
repeat
{
        status[i] := stage1;
        wait until(∀j ∈ HPP(i), stage1[j] = out)
        status[i] := stage2;
}until (∀k ≠ i, status[k] ≠ stage2)
turn := i;
CS;
if(i = n) then turn := 1; else turn := i+1;
status[i] := out;
```

Figure 3. Knuth's Algorithm.

3.4 Lamport's Algorithm (Bakery Algorithm)

Lamport's algorithm is based upon the service strategy commonly used in bakeries (hence the name Bakery Algorithm). A process chooses a token number upon entering the competition for shared resource. The holder of the lowest token number is the next one to be served. If two processes choose the same token number due to concurrency, then the process with lowest id goes first. The algorithm is given in Figure 4.

Process i:

```
choosing[i] := 1;
number[i] := 1+maximum(number[1], ..., number[n]);
choosing[i] := 0;
for j := 1 to n
{
        wait until (choosing[j] = 0)
        wait until ((number[j] = 0) or ((number[j],j) ≥ (number[i],i)))
}
CS;
number[i] := 0;
```

Figure 4. Lamport's Algorithm.

3.5 Peterson's Algorithm

The basic idea behind Peterson's algorithm is that each process has to pass through *n-1* stages to access the shared resource. These stages are designed to *block* one process per stage so that after *n-1* stages only one process will be eligible to access the shared resource. A process can move to next stage only if it is either pushed by some other process or all other processes are in stages below its own. A non-competing process is considered to be at stage 0. The algorithm is given in Figure 5.

Process i:

```
for j := 1 to n-1
{
        Q[i] := j;
        TURN[j] := i;
        Wait until(∀k ≠ i, Q[k] < j) or (TURN[j] ≠ i)
}
CS;
Q[i] := 0;
```

Figure 5. Peterson's Algorithm.

4. Analysis

In this section, we briefly discuss the observations from the analysis of the above five algorithms with respect to generalization, non-circularity, and bounded bypass properties. We first explain the concept/property and then present our observation.

4.1 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 1: Generalization is passing from the consideration of a restricted set to that of a more comprehensive set *containing* the restricted one. - George Polya[13].

Definition 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[5].

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

Dijkstra has never claimed his algorithm as a generalization of Dekker's algorithm. However, many researchers and most text books on this subject refer so. Even the recent references [1,2] confirm such a myth that Dijkstra's algorithm is a generalization of Dekker's algorithm. The differences between Dekker's algorithm and Dijkstra's algorithm are significant even when *n=2* and that violate the basic requirement of generalization, which we have discussed above.

Theorem 4.1.1 *Starvation is not possible in Dekker's algorithm.*

Proof: In Dekker's algorithm, the tie is broken by favoring the process which accessed the CS "least recently". This policy allows the *turn* to go between the competing processes alternatively. Therefore, no starvation is possible.

Theorem 4.1.2 *Dijkstra's algorithm is susceptible to starvation, even when n = 2.*

Proof: In Dijkstra's algorithm, the tie is broken by favoring the process which accessed the CS "most recently". This policy might lead to starvation if the process, which holds the current *turn*, is continuously interested in accessing the CS.

From Theorems 4.1.1 & 4.1.2, it is clear that Dijkstra's algorithm cannot be considered as a generalization of Dekker's algorithm.

4.2 Non-circularity Property

The property of not assuming *atomicity* on *read* and *write* operations has been referred in the literature as *non-circularity property*. The mutual exclusion algorithms with non-circularity property are called *non-atomic* algorithms. It was widely believed that the *read* and *write* operations on an individual memory word are atomic (mutually exclusive) is a required assumption for any mutual exclusion algorithm of shared memory systems. This belief was observed as incorrect by Lamport in [10], referring that Bakery algorithm does not require such assumption. Recently, in [1], bakery algorithm has been credited as the first non-atomic algorithm to assure mutual exclusion. However, it is interesting to know that Dekker's algorithm and the algorithm by Knuth, proposed more than eight years before Bakery algorithm was published, are indeed non-atomic algorithms assuring safety and liveness properties.

Consider Dekker's algorithm. In that, a process writes its status value, to indicate its competition for the CS, before reading the status value of the other process. Since the operations within a process are sequential, at least one write should precede both reads. That means, at least one process can observe the status value of other process correctly, and therefore it cannot come out of the while-loop. Thus, mutual exclusion will be preserved always in Dekker's algorithm irrespective of whether the operations are atomic or not. The similar argument is valid for Knuth's algorithm.

Theorem 4.2.1 *Atomicity assumption is not required to assure the safety property of the algorithms by Dekker and Knuth.*

4.3 Bound on Bypass

When several processes are competing for a shared resource, some later processes may *overtake* or *bypass* other earlier processes. This may happen due to different execution speeds of the processes. The bound on the number of bypass over a process in an algorithm indicates the *fairness* assurance of that algorithm. Peterson has never claimed that his algorithm assures bounded bypass. But, independently, the bound of Peterson's algorithms has been computed by different researchers as follows: Raynal[14] computed the bound as *n(n-1)/2*; Kowaltowski and Palma[8] claimed it as *n-1*; and Hofri[6] derived it as *n-1*.

Consider the following scenario. The processes p1, p2, and p3 are currently competing for the CS.

- Process p1 starts first, sets Q[p1] := 1,
- p3 starts and sets TURN[1] := p3,
- p2 sets TURN[1] := p2 and so p3 is pushed,
- since the condition ($\forall$ k $\neq$ p2, Q[k] < 1) or (TURN[1] $\neq$ p2) is not true, p2 is blocked at stage 1,
- p3 crosses stage 1 to stage 2,
- since the condition ($\forall$ k $\neq$ p3, Q[k] < j) is true, for j $\geq$ 2, it keeps proceeding further, enters the CS, and completes its CS execution,
- p3 starts competing again for the CS, and sets TURN[1] := p3,
- the condition (TURN[1] $\neq$ p2) becomes true, that is, p2 is unblocked and p3 is blocked at stage 1,
- p2 moves up all the way, enters and leaves the CS, starts competing again, and sets TURN[1] := p2,
- this time p2 gets blocked and p3 is unblocked, at stage 1, and
- p2 and p3 can overtake p1, alternately, several times until p1 sets TURN[1] := p1.

Therefore, there is no bound on the number of possible bypasses in Peterson's algorithm.

5. Conclusion

Concurrent programs are extremely hard to design and notorious for subtle errors. Slips are often possible while characterizing, designing, and proving the properties of concurrent programs. In this context, precise understanding of the concepts and ideas are extremely important and any misleading interpretations or references about popular algorithms will only add further complexity to the subject matter.

As a result of these observations, we feel that, (i) any error or inaccuracy observed should be disseminated to the scientific community as soon as they are identified. This may increase the clarity of the subject matter and avoid cascading or perpetual errors in concurrent programs; (ii) a systematic approach (structured software engineering techniques and tools) to design concurrent programs would help to alleviate the intricacy involved in it, and (iii) more importantly, it is essential to understand and accept that a different mindset[15] is required to design 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, 16:75-110, 2003.
3. E.W. Dijkstra, Solution of a Problem in Concurrent Programming Control, CACM, Vol.8(9):569, 1965.
4. E.W. Dijkstra, Cooperating Sequential Processes (Techniche Hogeschool, Eindhoven, 1965). Reprinted in: F. Genuys (ed.), Programming Languages, Academic Press, 43-112, 1968.
5. Albert Einstein and Leopold Infeld, The Evolution of Physics – The Growth of Ideas from Early Concepts to Relativity and Quanta, Cambridge University Press, 1971.
6. M. Hofri, Proof of a Mutual Exclusion Algorithm - A `class'ic example, ACM SIGOPS OSR 24(1):18-22, 1990.
7. D.E. Knuth, Additional Comments on a Problem in Concurrent Programming Control, CACM 9(5):321-322, 1966.
8. T. Kowalttowski and A. Palma, Another Solution of the Mutual Exclusion Problem, IPL (19), 3, 145-146, 1984.
9. L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem, CACM 17(8):453-455, 1974.
10. L. Lamport, The Mutual Exclusion Has Been Solved, CACM 34(1):110, 119, 1991.
11. L. Lamport, Description about Bakery Algorithm – In My writings http://research.microsoft.com/users/lamport/pubs/pubs.html.
12. .L. Peterson, Myths about the Mutual Exclusion Problem, IPL 12(3):115-116, 1981.
13. G. Polya, How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 1973.
14. M. Raynal, Algorithms for Mutual Exclusion, MIT press, 1986.
15. Resnick, Beyond the Centralized Mindset, Journal of the Learning Sciences, 5(1):1-22, 1996.