Proc. 1988 International Conference on Supercomputing, St. Malo, France, pp. 621-626, July 1988.

A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait

by
Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA

Abstract

A new simple solution to the Lamport's concurrent programming problem is presented. The algorithm uses five distinct values of shared memory per process. The shared values can be stored either in a single variable or in three one-bit boolean variables assigned to each process. The algorithm exhibits strong fairness property by enforcing the linear wait. It can be made immune to two types of errors typical to VLSI chip based multiprocessor systems: process failures and restarts, and read errors occurring during writes.

The algorithm requires a small number of writes to shared memory. At most $4\times p-\left\lceil \frac{p}{n} \right\rceil$ writes are needed for p entries to critical section by n competing processes. The algorithm's scheme is similar to that of Morris's solution to the mutual exclusion based on three weak semaphores.

1.Introduction

Mutual exclusion is one of the most fundamental problems that involve controlling parallelism. The issue here is to limit parallelism of a number of concurrent processes at certain instances of their execution. The code executed in those instances typically contains accesses to memory locations, or to some other resources that permit only one process at a time to access them. Such code is often referred to as a *critical section* or a *critical region* [Dijkstra, 1965]. Processes can proceed in parallel outside their critical sections but only one process at a time can execute its critical section.

To provide mutual exclusion in uniprocessor systems, it is often enough to disable interrupts when a process is in its critical section. Such solution is efficient only if critical sections are short. Otherwise the system response time would degrade, or disabled interrupts could be mishandled. The other limitation of this technique is that disabling/enabling interrupts cannot be made available to user programs in most systems.

In multiprocessors with shared memory the technique using special test-and-set instruction can be effective. However, it requires synchronized accesses from all processes to the memory. In a multiprocessor, multiport memory system it is impossible to create a test-and-set by merely controlling the access cycle of a single processor [Ferguson, 1984; Peterson, 1983]. For example, there always is some time delay in a clock pulse across the chip. Consequently, on a VLSI chip with thousands of processors, the processors cannot run on the same clock. Growing popularity of parallel and distributed architectures has led to renewed interest in algorithmic solutions to the mutual exclusion problem [Davidson, 1987; Ferguson, 1984; Peterson, 1983; Raynal, 1986].

The question of how to implement mutual exclusion algorithmicly has been studied extensively [Dijkstra, 1965; Knuth, 1966; Eisenberg and McGuire, 1972; Raynal, 1986]. In [Lamport, 1974], Lamport presented a new extended definition of the mutual exclusion problem. The new definition takes into account possible process failures and imposes restrictions on use of shared memory. In this new form, the problem became even more relevant to VLSI chip based multiprocessor systems, in which nonuniform conditions in the chip's wafer result in varying reliability of individual processors.

Acknowledgment: This work was partially supported by the Office of Naval Research through contract #N00014-86-K-0442 and by the Army Research Office through contract #DAAL03-86-K-0112.

Lamport's Bakery Algorithm [Lamport, 1974] uses shared variables of unbounded size and is not immune to infinite failures. Bounded-size solutions that are immune to infinite failures were published in [Katseff, 1978; Peterson, 1983; Rivest, 1976]. All these solutions, however, require unbounded number of writes to shared memory.

An important characteristic of any Lamport's concurrent programming problem solution is the maximum number of distinct values of shared memory used by each process. Peterson presented the solution that uses only four distinct values but which does not satisfy the strong fairness condition (the solution supports quadratic and not linear delay) [Peterson, 1983]. The algorithm given here is not only simpler than the one presented by Peterson, but it also enforces linear wait.

The presented algorithm uses just five distinct values of shared memory per process. These values can be packed into a single variable or represented by three boolean variables. The implementation with three boolean variables can be made immune to the following two types of errors considered by Lamport:

(1) unbounded (possibly infinite) number of process failures and restarts, and

(2) read errors occurring during writing of a shared variable.

Another important characteristic of the solution is the number of required writes to the shared memory. If the waiting processes are suspended, and not cycling in a busy wait, then the number of writes dictates memory traffic as well as the number of wake-ups of suspended processes. The described algorithm requires at most $4 \times p - \left\lceil \frac{p}{n} \right\rceil$ writes for p entries to the critical section made by n competing processes.

The algorithm's scheme is similar to that of Morris's solution to the mutual exclusion based on three weak semaphores [Morris, 1979].

2. The Problem Statement

The Lamport's concurrent programming problem has been fully defined elsewhere [Lamport, 1974], so only a general description is given here. There are n (n>1) processes that are numbered from 0 to n-1. The processes are executing independently of each other, possibly on different processors. The code of each process is divided into two parts: a *critical section,* which typically contains accesses to some resources, and a *noncritical section.* There is no assumption about the rate at which processes execute. However, each process in a critical section makes a finite progress. This means that a finite, but possibly unbounded, amount of time elapses between the execution of individual instructions of code. Finally, a process entering its critical section is assumed to leave it after a finite amount of time.

Processes start execution at a specified location in their noncritical part of code, with all variables set to initial values. Each process can enter the critical section any number of times. Processes can communicate with each other through the *shared memory.*

Algorithmic solutions of this problem consists of two sections of code which surrounder the critical section in each process. The first section is executed before critical section and is called a *prologue.* The second section is executed after critical section, and is called accordingly an *epilogue.* The assumption about finite progress in execution of the critical section is extended to the prologue and epilogue as well. However, this does not mean that a process which started to execute its prologue or epilogue, has to leave any of them in a finite time. In other words, an infinite looping is excluded by assumption, and should be avoided through proper design of the prologue and epilogue. We are interested in a uniform solution, in which all processes execute the same prologue and epilogue.

There are four properties required from the correct solution.

1. Mutual exclusion: There will be at most one process executing the critical section at a time.
2. Freedom from deadlock: The critical section will not become inaccessible to all processes. This means that if a number of processes attempts to execute their critical sections, then after finite amount of time some process will be allowed to do so.
3. Fairness (freedom form starvation): No process will be denied entry to its critical section forever. Thus, a process requesting an entry to its critical section will enter it after waiting for a finite amount of time. The stronger fairness property requires that no process can enter its critical section twice while another process is waiting (linear wait).
4. Robustness: The solution should be immune to the following two types of failures:

   (1) Process failure: a process may repeatedly fail and restart. However, process failing in the critical section, prologue or epilogue is assumed to leave the respective section of code and reset all its variables to their initial values.

   (2) Read errors during writes (flickering bits): when a process writes a new value to a shared variable, a sequence of reads may return any sequence of the old and new values.

The robustness requirement implies that global variables cannot be used, for the process associated with a global variable may fail and take the variable with it. Similarly, in general no process can rely on another's variables.

Thus, a robust algorithm shall use only so called process specific shared variables [Raynal, 1986].

A process specific shared variable can be written only by one process ("owner" of that variable). It may be read by all processes. As pointed out in [Peterson, 1983], read errors during writes can easily occur when two processors running on different clocks communicate. The sum of pulses from the two processors may create so called runt pulse, which causes the read to return a random value. Thus, the algorithm robustness is an essential requirement for solutions targeted to VLSI chip based multiprocessor systems. Since space is at premium in a chip, then minimizing amount of needed shared memory is also of importance in such systems.

3. The Algorithm

The idea behind the algorithm is simple. The prologue contains a waiting room with two doors. At the course of the execution, processes assume different states in relation to the entry to the critical section (comp. Figure 1). All processes requesting entry to the critical section at roughly the same time gather first in the waiting room. Then, when there are no more processes requesting entry, waiting processes move to the end of the prologue. From there, one by one, they enter their critical sections. Any other process requesting entry to its critical section at that time has to wait in the initial part of the prologue (before the waiting room).

In the waiting room of the prologue, always one door is opened while the other is closed. Initially, the door_in is opened. Each process passing through this door checks whether there is any other process intending to enter critical section. If there is, than the door_in remains opened and the passing process moves to the waiting room. Otherwise, it closes door_in and opens door_out. The process which has the lowest order number among processes that passed the door_in enters its critical section. The door_in is opened (and the door_out is closed) when the last process from those that already passed the door_in leaves the epilogue.

One process specific shared variable, called flag, describes the current state of its owner process. This variable assumes one of the following five values:

0 - denoting that the owner process is in the noncritical section,

1 - indicating that the owner process wants to enter its critical section (declaration of intention),

2 - showing that the owner process waits for other processes to get through the door_in,

3 - denoting that the owner process has just passed through the door_in,

4 - indicating that the owner process has crossed the door_out.

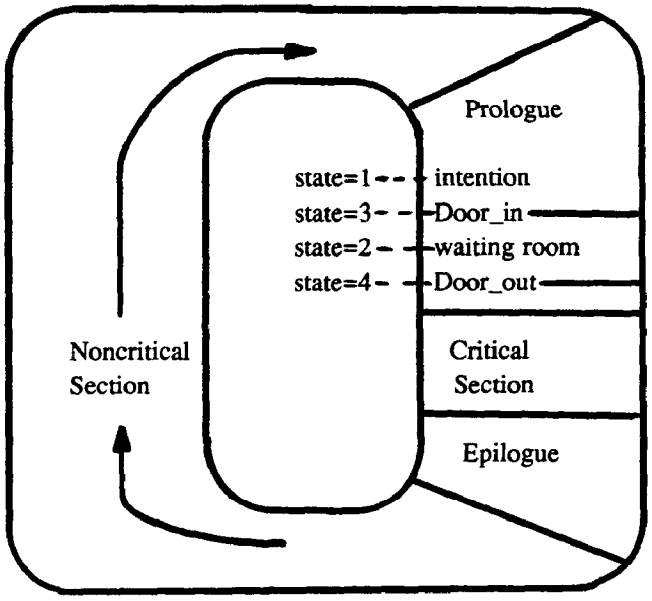

Figure 1. Scheme of Process States During Execution

At any time only one process can have the lowest order number in a set of processes which passed through the door_in, so the mutual exclusion is enforced. If a process overtakes the other process in entering its critical section, this process will not be able to pass through the door_in until all the processes it overtook leave their critical sections. Thus, the linear wait is also enforced. Finally, no process will wait in the waiting room forever, for in a group of processes which passed through the door_in there will always be a process that bypasses state 2 and changes its state from 3 to 4 directly. This is the process which was the last one to set its state to 3 in this group of processes. Once the state 4 has been reached by a process in this group, it will be maintained by at least one process in this group all the time, until all processes in the group leave the epilogue. The detailed algorithm is shown in Figure 2.

In the algorithm, the condition for closing the door_in and opening the door_out is $\forall$ j: flag[j]$\neq$1. The condition for closing the door_out and opening the door_in is $\forall$ j: flag[j]$\neq$4. Each process passing through the door_in closes it momentarily to check if it should close it permanently (state 4) or keep opened (state 2).

Note that step E0 is needed to keep the door_out opened for processes still in the waiting room. We can move this statement to the prologue, just after the statement P30, and the algorithm still will be correct. Doing that in the epilogue increases efficiency of the solution, because a process can enter its critical section without waiting for any other process to get out of the waiting room.

For each process P_i,$0 \le i \le n-1$, the prologue and epilogue are:
specific shared flag **in** 0..4;
local integer j **in** 0..n-1;

P10: flag[i]:=1;
P11: **wait until** $\forall$ j: flag[j]<3;
P20: flag[i]:=3;
P21: **if** $\exists$ j: flag[j]=1 **then begin** flag[i]:=2;
P22: **wait until** $\exists$ j: flag[j]=4; **end;**
P30: flag[i]:=4;
P31: **wait until** $\forall$ j<i: flag[j]<2;

Critical Section

E0: **wait until** $\forall$ j>i: flag[j]<2 & flag[j]>3;
E1: flag[i]:=0;

Figure 2. The Algorithm

Lemma 1. The algorithm preserves mutual exclusion.

Proof. For a process to enter its critical section, it must first set its flag[i] to 4, and then not see any lower numbered flag set to the value bigger then 1. If for some j<i, the processes P_i and P_j were both in their critical sections, then at the moment the process P_i entered its critical section, flag[j] was 0 or 1 (comp. statement P31).

Process P_i could set its state to 4 either directly from statement P21 or indirectly through a wait in statement P22. In the former case the process P_j had to be in state 0 at the moment the process P_i was executing statement P21 (comp. condition of P21). At that time flag[i] had been already set to 3 to be later changed to 4. Thus, process P_j cannot pass beyond the statement P11 until the process P_i leaves its critical section (comp. condition of P11).

The only remaining possibility is that the process P_j was in state 1 when the process P_i was executing statement P21. When the process P_i passed statement P11, no process was in state 4. Let's consider the first moment after that in which any process reaches state 4. At that moment the process P_j has to be in state 0, and therefore it again cannot pass beyond the statement P11 until the process P_i leaves its critical section (the wait in E0 ensures that the process first to reach state 4 maintains this state until P_i changes its state to 4)□

Lemma 2. The algorithm prevents deadlock.

Proof. The only possible transitions of the process states in the prologue are: 1 to 3, 3 to 2, 3 to 4, and 2 to 4. Thus, if the deadlock were to occur, then after the finite amount of time we would have a group of processes with their flags set to values greater than 0, and not changing those values. Let's assume then, that we have such a group of waiting processes.

The flag value equal to 3 is temporarily only and a process always changes this value either to 2 or 4 in a finite time (thanks to our assumption about finite progress in prologue and the statement P21). Therefore no process in the waiting group can have flag set to 3.

If any process in the waiting group would have its flag set to 4, then no process in this group could have its flag equal to 2 (such a process would be able to pass the statement P22, and change the value of its flag to 4). Since in the previous paragraph we showed that no process can have its flag set to 3, then it follows that the processes in the waiting group have their flags equal to 4 or 1. This means that the process with the highest order number among those with flags equal to 4 can enter its critical section - a contradiction. Thus, no process in the waiting group can have its flag set to 4.

If any process in the waiting group would have its flag equal to 1, then at least one waiting process would have the flag equal to 4. Otherwise, the process with the flag equal to 1 would be able to pass the statement P11 and change its flag to 3. Since we already showed in the previous paragraph that no process can have its flag set to 4, than the processes in the waiting group cannot have flags equal to 1 either.

The only remaining possibility is that all processes in the waiting group have their flags equal to 2. Let's consider the waiting process that was the last to change its flag from 1 to 3. Executing the statement P21, this process could not see processes with flags equal to 1, so it had to change its flag to 4 and not to 2, as we assumed. This final contradiction shows that the group of processes waiting forever cannot exist. □

Lemma 3. The algorithm exhibits linear wait property.

Proof. Suppose that the process P_j set its flag to 1 not earlier than the process P_i, but after that it entered its critical section before P_i did. At the moment the process P_j changed its flag's value from 1 to 3, no process could have the flag set to 4. Therefore, either the process P_i had already its flag set to 3, or no process, including P_j, could set the flag to 4, until the process P_i did so. It follows, that at the moment the process P_j set its flag to 4, the process P_i had already its flag greater than 1. By the time the process P_j left its critical section, the process P_i had to set its flag to 4, thanks to the statement E0. Consequently, until the process P_i does not leave its critical section, the process P_j cannot get further then the statement P11 in its subsequent request to enter critical section. □

The following corollary immediately follows from the proved lemmas 1-3:

Corollary. The presented algorithm solves the mutual exclusion problem and exhibits the strong fairness property (linear wait).

5. Implementation

The described algorithm can be implemented using three boolean variables: intent, door_in and door_out. The values of this variables code the values of the corresponding flag as follows:

Coding of the flag values			
flag	intent	door_in	door_out
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	1	1	1

The essential feature of this coding is that almost all changes of the flags' values in the prologue require writing a new value to just one boolean variable. Indeed, the changes of flag values in the prologue call for the following assignments:

1. 0 to 1 requires intent:=true,
2. 1 to 3 calls for door_in:=true,
3. 3 to 2 requires intent:=false,
4. 3 to 4 calls for door_out:=true, and
5. 2 to 4 can be made in two steps: intent:=true (it is like a transition from 2 to 3) and then door_out:=true. Note, that transition 2 to 3 followed by 3 to 4, is in the described algorithm functionally equivalent to direct transition from 2 to 4.

It is easy to verify that this solution is immune to the bit flickering. For example, if the bit flickering occurs in case 1, the process reading the value of intent may simply close the door_in for the process currently writing its intent. In case 2, the reading process may be able to cross the door_in. In case 3, the reading process may be delayed at the door_in, etc.

To make the implementation immune to process failures and restarts in the prologue and epilogue, the following two modifications has been made in the described algorithm.

1. It is necessary to prevent deadlock from occurring when the process which is the last to change its flag from 1 to 3, fails before setting its flag to 4. In such case, processes waiting with the flag set to 2 could not leave the waiting room. The solution here is to extend the condition for leaving the wait in the statement P22 by boolean term $\forall$ j: flag[j]≠1 and to send processes satisfying this condition back to the statement P20.
2. In the described algorithm, the failing process may overtake other processes to the entry to its critical section after each failure and restart. This could happen, if the failing process is the only one with the flag equal to 4, and it fails in the epilogue. The required modification is to move the statement E0 from the epilogue to the prologue (just after the statement P30).

The modified algorithm written in C-like syntax is shown in Figure 3.

To make the algorithm immune to infinite number of failures and restarts, it is necessary to impose a condition on the restart of the failured process. Otherwise, a process can fail and restore constantly, and after each restart, it can immediately request entry to its critical section. If the failing process is fast enough in doing that, then all other processes waiting for entries to their critical sections may constantly see intent set to 1, so these processes would wait forever in the waiting room. To prevent this from happening it is sufficient to request that the restarted process waits until $\forall$ j: flag[j]≠2. This condition can be implemented by the following loop:

R0: j=0; while(j<n) if (!intent[j] & door_in[j]) j=0; else j++;

Restart

For each process P_i,0≤i≤n−1, the prologue and epilogue are:

```
specific boolean intent, door_in, door_out = false;
local integer j=0; boolean f=true;

P10: intent[i]=true;
P11: while(j<n) if (intent[j] & door_in[j]) j=0; else j++;
P20: door_in[i]=true;
P21: j=0; while ((!intent[j] | door_in[j]) & j<n-1) j++;
     if (intent[j] & !door_in[j]) { intent[i]=false;
P22:    while (!door_out[j] & (j<n-1 |
                f=intent[j] & !door_in[j] | f & j>0))
                if (j==n-1) j=0; else j++;
        intent[i]=true;
        if (!door_out[j]) goto P21; }
P30: door_out[i]=true;
E0:  j=i+1; while(j<n)
        if (door_in[j] & !door_out[j]) j=i+1; else j++;
P31: j=0; while(j<i) if (door_in[j]) j=0; else j++;

Critical Section

E1: intent[i]=false; door_in[i]=false; door_out[i]=false;
```

Figure 3. The Implementation of the Modified Algorithm

The original algorithm is deadlock free, therefore if there is a group of processes with their flags equal to 2, then after a finite amount of time they will start entering their critical sections. At that time the only flag values in existence will be 0, 1, and 4. Thus, no process will wait for a restart forever.

5. Conclusion

A robust, fair mutual exclusion algorithm that can be made immune to two types of failures has been presented. This algorithm uses just five distinct values of shared memory per process. These values may be stored either in a single variable or in three one-bit boolean variables per process. In the latter implementation, processes may repeatedly fail and restart, and read errors may occur during writes. The algorithm uses just one more value per process than the solution proposed in [Peterson, 1983]. However, the presented algorithm exhibits stronger fairness property and is much shorter. Another advantage of the presented algorithm is the small number of changes of shared values made during each prologue execution. The algorithm requires at most $4\times p-\left\lceil \frac{n}{p} \right\rceil$ writes of new values to shared memory per p entries to the critical section. Any change in conditions for all waits in the prologue and epilogue require a new write to be done. Therefore the small number of writes to shared memory can make an implementation of waiting through process suspension more efficient than implementations based on busy wait.

References

1. Davidson, C.M., A Note on Concurrent Programming Control, IEEE Transaction on Software Engineering, vol. SE-13, no. 7, July, 1987, pp. 865-866.
2. Dijkstra, E.W. Solution to a problem in concurrent programming control, Communication of the ACM, vol. 8, no. 9, September, 1967, p. 569.
3. Eisenberg, M.A., and McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem, Communication of the ACM, vol. 15, no. 11, November, 1972, pp. 999.
4. Ferguson, M.J. Multiaccess in a Nonqueueing Mailbox Environment, IEEE Transaction on Software Engineering, vol. SE-10, no. 3, May, 1984, pp. 237-243.
5. Katseff, H.P. A new solution to the critical section problem, in Conference Record of the Tenth Annual ACM Symposium on Theory of Computing, San Diego, CA, May 1-3, 1978, pp. 86-88.
6. Knuth D.E., Additional comments on a problem in concurrent programming control, Communication of the ACM, vol. 9, no. 5, May, 1966, p. 321-322.
7. Lamport, L. A new solution of Dijkstra's concurrent programming problem, Communication of the ACM, vol. 17, no. 8, August, 1974, pp. 453-455
8. Morris, J.M. A starvation-free solution to the mutual exclusion problem, Information Processing Letter, vol. 8, no. 2, 1979, pp. 76-80.
9. Peterson, G.L. A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables, ACM Transactions on Programming Languages and Systems, vol. 5, no. 1, January 1983, pp. 56-65.
10. Raynal, M. Algorithms for Mutual Exclusion, The MIT Press, Cambridge, Massachusetts, 1986.
11. Rivest, R.L., and Pratt, V.R. The mutual exclusion problem for unreliable processes; Preliminary Report. Proc. 17th Annual Symposium on Foundations of Computer Science, TX, 1976, pp. 1-8.