A Survey of Mutual-Exclusion Algorithms for Multiprocessor Operating Systems

Lawrence Kesteloot

January 20, 1995

1 Introduction

The problem of *mutual-exclusion* is that of guaranteeing that certain sections of code (*critical sections*) will not be executed by more than one process simultaneously. These sections of code usually access shared variables in a common store or access shared hardware. The actual contents of the critical section is immaterial to the problem of mutual-exclusion, although there is usually an assumption that the section is fairly short.

The standard solution to kernel-level mutual-exclusion in uniprocessor systems is to momentarily disable interrupts to guarantee that the process accessing the sensitive data will not be pre-empted before the access has been completed. This solution is not available for multiprocessor systems, since processes on these are truly concurrent. This paper surveys the published solutions to this problem and their properties. We also review some current work on the subject and other possible solutions to concurrent access of shared variables.

2 Terms and Issues

A critical section of code is framed by an *entry section* at the beginning and an *exit section* at the end; these sections act to grab and release the "lock" on that section. To simplify the model, we will assume that there is only one critical section, and not multiple sections of code that have to be mutually exclusive. It is easy to see that this simplification does not weaken the model.

When discussing distributed and concurrent algorithms, there are two kinds of properties with which we must be concerned. A *safety property* is one which guarantees that a bad thing will not happen. These are fairly easy to prove about algorithms because only the static state of the system at any particular time is taken into account. The other kind is a *progress property*, which guarantees that a good thing will eventually happen. This is much more difficult to prove since time must be taken into account. Proving the correctness of programs is essential with concurrent algorithms because our intuitive notion of program flow is violated by the fact that the value of a variable may not be the same from one statement to the next, even if the process we are tracing does not modify it. Only through formal proof techniques can we be guaranteed that the above properties will be satisfied.

With the problem of mutual exclusion, one safety property is that of *mutual exclusion*; that is, not more than one process should have its program counter (PC) in the critical code at the same time. To prove this, it suffices to show that one process's PC cannot leave the entry section while other process's PC is between the entry section and the exit section. Another desirable safety property is that of *freedom from deadlock*. This property guarantees that not all processes will be stuck in the entry section waiting for another process to enter the critical section.

There are two progress properties that are desirable with mutual-exclusion algorithms. The first is that of *freedom from livelock*. This property can be phrased as, "If some process wants to enter the critical section, then *some* process will eventually enter the critical section." The other progress property, *freedom from starvation*, is stronger than the first and is phrased as, "If some process wants to enter the critical section, then *that* process will eventually enter the critical section." The latter property implies the former, but is more difficult to guarantee. Notice also that freedom from livelock implies freedom from deadlock.

The terms *process* and *processor* are used interchangeably in this paper, since we assume that only one kernel-level process is running on each processor. Multiple processes on a single processor can be handled in a traditional way (*e.g.*, by disabling interrupts).

Lastly, throughout most of this paper, we assume that no processor will halt (*i.e.*, crash) while inside the critical section. This restriction is necessary because it is generally not possible for a process waiting on a lock to know whether the process that has the lock has crashed or is simply slow in its use

of the shared resource. Halting in the critical section, then, would cause all other processes to block indefinitely in the entry section. The last section of this paper investigates algorithms that are resilient to arbitrary processor crashes.

3 History of Solutions

The problem of multiprocessor mutual-exclusion was first proposed in 1962 by T. Dekker, but no correct solution for more than two processes was published until 1965, when Edsger Dijkstra wrote his one-page article [2] giving a working but complicated solution to the problem. His solution guaranteed mutual exclusion, freedom from deadlock, and freedom from livelock, but allowed the possiblity of one process being forever stuck in the entry section while other processes are allowed into the critical section. This paper was quickly followed by Donald Knuth's equally complicated solution [4], which did guarantee freedom from starvation.

Both of the above algorithms had the fault that if a single processor were to halt (*i.e.*, crash) at any point in the entry section, the whole system might block indefinitely. Also, both solutions assumed atomic reads and writes to memory; that is, they assumed that all reads and writes to single memory locations will be non-overlapping, presumably arbitrated by hardware. This assumption is often not valid, and we must deal with the possiblity that a read may overlap a write, thus returning invalid or partial results, or that several writes may overlap, thus writing invalid or partial data.

Eight years after Knuth's article, Leslie Lamport published a solution [5] which is not only simpler that the first two, but which both allows any processor to halt anywhere in the entry section and allows a read to return any arbitrary value if it overlaps a write. Lamport's algorithm also had the new property that processes were served in a first-come first-served order, which Dijkstra's and Knuth's algorithms did not guarantee.

Knuth's algorithm was basically analoguous to a bakery, where customers take a number when they walk in the door and wait for their number to be called. In the algorithm, each process that wanted to enter the critical section would select a number that was one greater than the maximum number chosen by the other processes, and the process with the lowest number would be allowed in.

In 1981, seven years after Knuth's article, Gary Peterson published his now-famous algorithm [11] for two processes, and also gave an algorithm for more than two processes. The two-process algorithm was so simple that he felt that a proof of correctness was not necessary, and it is worth reproducing here:

```
/* Entry section for P1  */      /* Entry section for P2 */
Q1 := True;                      Q2 := True;
TURN := 1;                       TURN := 2;
wait while Q2 and TURN := 1;     wait while Q1 and TURN := 2;

/* Exit section for P1 */        /* Exit section for P2 */
Q1 := False;                     Q2 := False;
```

Both in this algorithm and in all of the previous ones, the processes busy-wait until they are allowed to enter the critical section. One might wonder why these algorithms busy-wait instead of simply context-switching to another ready process. At the beginning of this paper, we made the assumption that critical sections were usually very small; this would imply that any process busy-waiting on a lock would likely wait for a fairly short period of time. The overhead of context-switching is usually large enough that busy-waiting is more efficient. For critical sections that are longer, such as those with disk access, the above algorithms can be used as low-level building blocks for higher-level semaphores.

4 Beyond Simple Mutual Exclusion

With the problem of general mutual exclusion solved, some special cases were brought up. For example, consider the situation where the critical section is one where access (reading and writing) to a shared data structure is made. There will be some processes that are interested in writing to that structure, and some that are interested in reading. We certainly want to protect the data structure from concurrent writes, and we want to insure that reads don't happen in the middle of a write. But is it reasonable to forbid concurrent reads? In 1971, Courtois, Heymans, and Parnas [1] proposed two algorithms that allowed concurrent reads while ensuring that writes are exclusive from

each other and from reads. One solution attempted to have a minimal delay for reads, and the other to have a minimal delay for writes.

The first solution was quite simple and ingenious: simply treat all readers as one process. The first reader who wants entry must get the main semaphore. While at least one reader is in the critical section, other readers are allowed free entry and exit. The last reader to exit releases the semaphore and allows entry to a waiting writer. This solution has the fault that a writer may be blocked indefinitely while an infinite number of readers stream though the critical code. Their second solution reverses this by allowing readers to block indefinitely while writers get immediate access to the data.

When dealing with multiprocessor systems, it is important to re-evaluate the solutions that we use in uniprocessor systems. For example, mutual-exclusion is a reasonable solution in uniprocessor systems because only one process can execute at any one time, so mutual exclusion really only specifies their order of scheduling. However, on multiprocessor systems, mutual exclusion is an expensive operation, not only because of the overhead involved, but because one processor will sit idle while the other has access to the data. This should motivate some algorithms that allow readers and writers to access the data without forcing either to block. Even a partitial solution where only writers never block would be useful for large data structures where an update (write) might take some time and should be allowed immediately.

Leslie Lamport suggested an algorithm [7] which would always allow a single writer to write without blocking. The basic idea was to allow the writer to write at any time, and the reader to repeatedly read the data until it is sure that it has a valid copy. The writer should write data version numbers both before and after the update, and the reader should read them in reverse order and compare them. Pseudo-code may help clarify the algorithm:

```
/* Algorithm for writer */      /* Algorithm for reader */
v1 := v1 + 1;                   repeat temp := v2;
write data;                       read data;
v2 := v1;                       until v1 = temp;
```

This assumes that the version numbers themselves can be written and read atomically. If they cannot, then they must be written in such a way as to guarantee that the reader will get a copy of `v1` which is greater than or equal to the actual copy, and a version of `v2` which is less than or equal to

the actual copy. Lamport showed that if `v1` is written most significant digit first and read least significant digit first, and if `v2` is written and read in the opposite order, then the above propreties will hold and an atomic write is not necessary. (He does assume that, at some level, an atomic write is possible, even if this write is at the level of a single bit.)

When mutual exclusion is the only option, we are then faced with the potentially large delays incurred in the entry section, particularly in large distributed systems. It became the belief of operating system designers that the great majority of processes that entered the entry section were allowed into the critical section without delay; *i.e.*, that contention was very low. With this in mind, several algorithms were developed that tried to reduce the $O(N)$ access time of the early mutual exclusion algorithms. Leslie Lamport once again led the way with his "fast" mutual exclusion algorithm [8], which only required seven memory accesses in the event of no contention.

It was then noticed that most distributed systems were set up so that each processor could quickly access its local memory and slowly access other processors' memories. The interconnect network was usually shared by all processors, and remote spin-locking seriously affected communication among other processors, even those that were not attempting entry into any critical section. Attention turned to minimizing remote memory accesses, and Mellor-Crummey and Scott developed an algorithm [10] that had $O(1)$ remote references.

In the late 1980's, multiprocessor systems made from ordinary CPU's became popular, and since these CPU's often did not have atomic test-and-set operations, operating system designers were once again faced with having to write mutual-exclusion algorithms from atomic reads and writes. Yang and Anderson [14] developed a $O(\log_2 N)$ algorithm which only required atomic reads and writes. In fact, even under heavy contention, performance tests showed that it rivaled algorithms that used hardware-supported atomic operations.

The delay in the entry section is a particularly important issue in message-passing systems, and several papers have focused on trying to reduce this overhead. One of the firsts was Ricart and Agrawala's solution [12] which used $2 * (N - 1)$ messages: $N - 1$ messages to ask for permission and $N - 1$ to grant it. This algorithm required consent from all of the other processors. Thomas [13] recognized that only a majority of the processors needed to

consend, and modified the algorithm to reduce the number of messages by half. Maekawa followed with a $3\sqrt{N}$ solution [9] in which $\sqrt{N}$ groups of processors were created and each processor was in $\sqrt{N}$ groups; this was a variation on the old divide-and-conquer idea.

5 Wait Freedom

Wait freedom is the idea that we can build a shared data structure which can be accessed simultaneously by several processes without waiting for exclusive access. All processes are guaranteed to complete the access in finite time, regardless of the actions of the other processes. The name "wait-free" is misleading because it implies that the purpose is to have fast access. Wait freedom is not about speed, it is about resiliency. In the algorithms we covered above, if the process that has the lock somehow halts or crashes, all other processes waiting on the lock will wait indefinitely. Wait-free synchronization avoids this problem by removing the spin-locks and setting up the data structures in such a way that concurrent access will result in the proper results. It also allows maximum concurrency by avoiding needless waiting.

We can setup a hierarchy of wait-free constructs. At the lowest level, the bit can be accessed in a wait-free manner; the writer toggles it when it needs to be changed and the readers can read it at any time. Multi-bit registers can be built from 1-bit registers (naïvely using a unary encoding with 2^n bits), and multi-writer and multi-reader registers can be built from these. Many of these constructions were outlined by Lamport in [6].

At this point, we must ask ourselves several questions. What kind of power is available to us with these atomic registers? Can we achieve distributed consensus (*i.e.*, voting by several processors) using only atomic registers? What kind of hardware support would we like for a multiprocessor operating system? What kind of synchronization can we achieve using only atomic registers? In 1991, Maurice Herlihy wrote the definitive paper on wait-free synchronization [3]. In it, he outlined a ladder of synchronization power that can be achieved with different wait-free constructs, such as the ones described above.

For example, if only read/write registers are available, then we can only reach consensus between two processors. This is significant because if we want to write an operating system which is to support hardware with more

than two processors, then we cannot rely simply on read/write atomicity if we want to use wait-free constructs. The test-and-set atomic operation, suprisingly, is also limited to two-processor synchronization. Finally, the more powerful compare-and-swap atomic operation can achieve synchronization between any number of processors.

6 Conclusion

Access to shared data is an essential part of distributed and concurrent computing, and algorithms to arbitrate concurrent accesses are at the heart of distributed operating systems. The algorithm of choice depends heavily both on the hardware support and on the hardware configuration. If resiliency is desired, then wait-free constructions will guarantee that inopportune processor failures will not halt other parts of the systems.

References

[1] Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with "readers" and "writers". *Communications of the ACM 14*, 10 (October 1971), 667–668.

[2] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM 8*, 9 (September 1965), 569.

[3] Herlihy, Maurice. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 11*, 1 (January 1991), 124–149.

[4] Knuth, Donald E. Addition comments on a problem in concurrent programming control. *Communications of the ACM 9*, 5 (May 1966), 321–322.

[5] Lamport, Leslie. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (August 1974), 453–455.

[6] Lamport, Leslie. On interprocess communication, Part II: Algorithms. *Distributed Computing 1*, (1986), 86–101.

[7] Lamport, Leslie. Concurrent reading and writing. *Communications of the ACM 20*, 11 (November 1977), 806–811.

[8] Lamport, Leslie. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems 5*, 1 (February 1987), 1–11.

[9] Maekawa, Mamoru. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems 3*, 2 (May 1985), 145–159.

[10] Mellor-Crummey, John M. and Scott, Michael L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (February 1991), 21–65.

[11] Peterson, G. L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (13 June 1981), 115–116.

[12] Ricart, Glenn and Agrawala, Ashok K. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM 24*, 1 (January 1981), 9–17.

[13] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems 4*, 2 (June 1979), 180–209.

[14] Yang, Jae-Heon and Anderson, James H. A fast, scalable mutual exclusion algorithm. *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York (August 1993), 171–182.