Concurrent Programming, Mutual Exclusion (1965; Dijkstra)

Gadi Taubenfeld, The Interdisciplinary Center, Herzliya, Israel

Synonyms: The critical section problem

1 Problem Definition

1.1 Concurrency, Synchronization and Resource Allocation

A *concurrent* system is a collection of processors that communicate by reading and writing from a shared memory. A distributed system is a collection of processors that communicate by sending messages over a communication network. Such systems are used for various reasons: to allow a large number of processors to solve a problem together much faster than any processor can do alone, to allow the distribution of data in several locations, to allow different processors to share resources such as data items, printers or discs, or simply to enable users to send electronic mail.

A *process* corresponds to a given computation. That is, given some program, its execution is a process. Sometimes, it is convenient to refer to the program code itself as a process. A process runs on a *processor*, which is the physical hardware. Several processes can run on the same processor although in such a case only one of them may be active at any given time. Real concurrency is achieved when several processes are running simultaneously on several processors.

Processes in a concurrent system often need to synchronize their actions. *Synchronization* between processes is classified as either cooperation or contention. A typical example for cooperation is the case in which there are two sets of processes, called the producers and the consumers, where the producers produce data items which the consumers then consume.

Contention arises when several processes compete for exclusive use of shared resources, such as data items, files, discs, printers, etc. For example, the integrity of the data may be destroyed if two processes update a common file at the same time, and as a result, deposits and withdrawals could be lost, confirmed reservations might have disappeared, etc. In such cases it is sometimes essential to allow at most one process to use a given resource at any given time.

Resource allocation is about interactions between processes that involve contention. The problem is, how to resolve conflicts resulting when several processes are trying to use shared resources. Put another way, how to allocate shared resources to competing processes. A special case of a general resource allocation problem is the *mutual exclusion* problem where only a single resource is available.

1.2 The Mutual Exclusion Problem

The *mutual exclusion* problem, which was first introduced by Edsger W. Dijkstra in 1965, is the guarantee of mutually exclusive access to a single shared resource when there are several competing processes [6]. The problem arises in operating systems, database systems, parallel supercomputers, and computer networks, where it is necessary to resolve conflicts resulting when several processes are trying to use shared resources. The problem is of great significance, since it lies at the heart of many interprocess synchronization problems.

The problem is formally defined as follows: it is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section* and *exit*. Thus, the structure of a mutual exclusion solution looks as follows:

loop forever
remainder code;
entry code;
critical section;
exit code
end loop

A process starts by executing the remainder code. At some point the process might need to execute some code in its critical section. In order to access its critical section a process has to go through an entry code which guarantees that while it is executing its critical section, no other process is allowed to execute its critical section. In addition, once a process finishes its critical section, the process executes its exit code in which it notifies other processes that it is no longer in its critical section. After executing the exit code the process returns to the remainder.

The Mutual exclusion problem is to write the code for the *entry code* and the *exit code* in such a way that the following two basic requirements are satisfied.

Mutual exclusion: *No two processes are in their critical sections at the same time.*

Deadlock-freedom: *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

The deadlock-freedom property guarantees that the system as a whole can always continue to make progress. However deadlock-freedom may still allow "starvation" of individual processes. That is, a process that is trying to enter its critical section, may never get to enter its critical section, and wait forever in its entry code. A stronger requirement, which does not allow starvation, is defined as follows.

Starvation-freedom: *If a process is trying to enter its critical section, then this process must eventually enter its critical section.*

Although starvation-freedom is strictly stronger than deadlock-freedom, it still allows processes to execute their critical sections arbitrarily many times before some trying process can execute its critical section. Such a behavior is prevented by the following fairness requirement.

First-in-first-out (FIFO): *No beginning process can enter its critical section before a process that is already waiting for its turn to enter its critical section.*

The first two properties, mutual exclusion and deadlock freedom, were required in the original statement of the problem by Dijkstra. They are the minimal requirements that one might want to impose. In solving the problem, it is assumed that once a process starts executing its critical section the process always finishes it regardless of the activity of the other processes. Of all the problems in interprocess synchronization, the mutual exclusion problem is the one studied most extensively. This is a deceptive problem, and at first glance it seems very simple to solve.

2 Key Results

Numerous solutions for the problem have been proposed since it was first introduced by Edsger W. Dijkstra in 1965. Because of its importance and as a result of new hardware and software developments, new solutions to the problem are still being designed. Before the results are discussed, few models for interprocess communication are mentioned.

2.1 Atomic Operations

Most concurrent solutions to the problem assume an architecture in which n processes communicate asynchronously via shared objects. All architectures support *atomic registers*, which are shared objects that

support atomic reads and writes operations. A weaker notion than an atomic register, called a *safe* register, is also considered in the literature. In a safe register, a read not concurrent with any writes must obtain the correct value, however, a read that is concurrent with some write, may return an arbitrary value. Most modern architectures support also some form of atomicity which is stronger than simple reads and writes. Common atomic operations have special names. Few examples are,

- *Test-and-set*: takes a shared registers r and a value *val*. The value *val* is assigned to r, and the old value of r is returned.
- *Swap*: takes a shared registers r and a local register ℓ, and atomically exchange their values.
- *Fetch-and-increment*: takes a register r. The value of r is incremented by 1, and the old value of r is returned.
- *Compare-and-swap*: takes a register r, and two values: *new* and *old*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise r is left unchanged and the value *false* is returned.

Modern operating systems (such as Unix and Windows) implement synchronization mechanisms, such as semaphores, that simplify the implementation of mutual exclusion locks and hence the design of concurrent applications. Also, modern programming languages (such as Modula and Java) implement the monitor concept which is a program module that is used to ensure exclusive access to resources.

2.2 Algorithms and Lower Bounds

There are hundreds of beautiful algorithms for solving the problem some of which are also very efficient. Only few are mentioned below. First, algorithms that use only atomic registers, or even safe registers, are discussed.

The Bakery Algorithm. The Bakery algorithm is one of the most known and elegant mutual exclusion algorithms using only safe registers [9]. The algorithm satisfies the FIFO requirement, however it uses unbounded size registers. A modified version, called the Black-White Bakery algorithm, satisfies FIFO and uses bounded number of bounded size atomic registers [14].

Lower bounds. A space lower bound for solving mutual exclusion using only atomic registers is that: any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers [5]. It was also shown in [5] that this bound is tight. A time lower bound for any mutual exclusion algorithm using atomic registers is that: there is no a priori bound on the number of steps taken by a process in its entry code until it enters its critical section (counting steps only when no other process is in its critical section or exit code) [3]. Many other interesting lower bounds exist for solving mutual exclusion.

A Fast Algorithm. A *fast* mutual exclusion algorithm, is an algorithm in which in the absence of contention only a constant number of shared memory accesses to the shared registers are needed in order to enter and exit a critical section. In [10], a fast algorithm using atomic registers is described, however, in the presence of contention, the winning process may have to check the status of all other n processes before it is allowed to enter its critical section. A natural question to ask is whether this algorithm can be improved for the case where there is contention.

Adaptive Algorithms. Since the other contending processes are waiting for the winner, it is particularly important to speed their entry to the critical section, by the design of an *adaptive* mutual exclusion algorithm in which the time complexity is independent of the total number of processes and is governed only by the current degree of contention. Several (rather complex) adaptive algorithms using atomic registers are known [1, 2, 14]. (Notice that, the time lower bound mention earlier implies that no adaptive algorithm

using only atomic registers exists when time is measured by counting *all* steps.)

Local-spinning Algorithms. Many algorithms include busy-waiting loops. The idea is that in order to wait, a process *spins* on a flag register, until some other process terminates the spin with a single write operation. Unfortunately, under contention, such spinning may generate lots of traffic on the interconnection network between the process and the memory. An algorithm satisfies local spinning if the only type of spinning required is local spinning. Local spinning is the situation where a process is spinning on locally-accessible registers. Shared registers may be locally-accessible as a result of either coherent caching or when using distributed shared memory where shared memory is physically distributed among the processors.

Three local-spinning algorithms are presented in [4, 8, 11]. These algorithms use strong atomic operations (i.e., fetch-and-increment, swap, compare-and-swap), and are also called *scalable* algorithms since they are both local-spinning and adaptive. Performance studies done, have shown that these algorithms scale very well as contention increases. Local spinning algorithms using only atomic registers are presented in [1, 2, 14].

Only few representative results have been mentioned. There are dozens of other very interesting algorithms and lower bounds. All the results discussed above, and many more, are described details in [15]. There are also many results for solving mutual exclusion in distributed message passing systems [13].

3 Applications

Synchronization is a fundamental challenge in computer science. It is fast becoming a major performance and design issue for concurrent programming on modern architectures, and for the design of distributed and concurrent systems.

Concurrent access to resources shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion *locks* (i.e., algorithms) are the de facto mechanism for concurrency control on concurrent applications: a process accesses the resource only inside a critical section code, within which the process is guaranteed exclusive access. The popularity of this approach is largely due the apparently simple programming model of such locks and the availability of implementations which are efficient and scalable. Essentially all concurrent programs (including operating systems) use various types of mutual exclusion locks for synchronization.

When using locks to protect access to a resource which is a large data structure (or a database), the *granularity* of synchronization is important. Using a single lock to protect the whole data structure, allowing only one process at a time to access it, is an example of *coarse-grained* synchronization. In contrast, *fine-grained* synchronization enables to lock "small pieces" of a data structure, allowing several processes with non-interfering operations to access it concurrently. Coarse-grained synchronization is easier to program but is less efficient and is not fault-tolerant compared to fine-grained synchronization. Using locks may degrade performance as it enforces processes to wait for a lock to be released. In few cases of simple data structures, such as queues, stacks and counters, locking may be avoided by using lock-free data structures.

4 Cross References

Registers, Self-stabilization, Wait-free computation.

5 Recommended Reading

In 1968, Edsger Wybe Dijkstra has published his famous paper "Co-operating sequential processes" [7], that originated the field of concurrent programming. The mutual exclusion problem was first stated and solved by Dijkstra in [6], where the first solution for two processes, due to Dekker, and the first solution

for n processes, due to Dijkstra, have appeared. In [12], a collection of some early algorithms for mutual exclusion are described. In [15], dozens of algorithms for solving the mutual exclusion problems and wide variety of other synchronization problems are presented, and their performance is analyzed according to precise complexity measures.

[1] J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proceedings of the 14th international symposium on distributed computing. Lecture Notes in Computer Science*, 1914:29–43, oct 2000.

[2] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 30:67–86, 2002.

[3] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] J.N. Burns and N.A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

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

[7] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968. Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[8] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computers*, 28(6):69–69, June 1990.

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

[10] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[11] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.

[12] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[13] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal Parallel Distributed Computing*, 18(1):94–101, 1993.

[14] G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 56–70.

[15] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education – Prentice-Hall, 2006. ISBN: 0131972596.