Shared Memory Synchronization

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel,
tgadi@idc.ac.il,
http://www.faculty.idc.ac.il/gadi/
September 1, 2008

Abstract. Synchronization is a fundamental problem in computer science. It is fast becoming a major performance and design issue for concurrent programming on modern architectures, and for the design of concurrent and distributed systems. In this survey, I have tried to gently introduce the reader to some of the most fundamental issues and classical results underlying the design of concurrent systems, so that the reader can get a feel of the nature of this exciting and important field. The topics covered are not an exhaustive survey of the field of synchronization algorithms and concurrent programming, but a subjective sampling of its highlights. All the results discussed here, and many more, are covered in details in [34].

1 Introduction

One of the major developments that is dramatically and irrevocably changing the computer landscape is the design of mainstream computers where a single computer has multiple processors. A processor is the "brain" of the computer, and mainstream computers are now built with multiple "brains".

These fundamental changes in computing architectures require a fundamental change in how applications are written for computers with multiple processors. Concurrency and synchronization are fundamental issues that are absolutely critical for the design of such applications. Much of the future of multiprocessor computers will be told by how well programmers can take advantage of concurrency.

There are two main technological underpinnings of the fascinating rapid developments of multiprocessor computers. The first is, of course, the advances in the design of faster hardware. The second is the development of efficient concurrent algorithms for supporting complex interactions between processors.

A concurrent algorithm is the recipe upon which a problem is solved, by more than one computing element. In 1968, Edsger Wybe Dijkstra, has published the paper that originated the field of concurrent programming [10]. In this survey, I have tried to gently introduces the reader to some of the principles and results underlying the design of such concurrent algorithms. All the results discussed here, and many more, are covered in details in [34].

Computers with Multiple Processors

A *processor* is a component in a computer that interprets and execute computer program instructions and processes data. Throughout the history of modern computing,

application developers have been able to rely on improved processor design to deliver significant performance improvements while actually reducing costs at the same time. That is, as processors got faster so did the applications. Unfortunately, increasing difficulty with heat and power consumption of the new smaller and faster processors along with the limits imposed by quantum physics has made this progression increasingly more difficult.

Until few years ago mainstream computers were built with a single processor only. This situation is changing fast, as it becomes more difficult to increase the (clock) speed of uniprocessor computers. Hence, all microprocessor companies have been forced to bet their futures on multiple processors (also called multicores[1]) which reside inside a single computer.

Most computers manufactures are now offering a new generation of multiprocessor computers where a single computer includes several processors all executing concurrently, and intricate and collaborate with one another. Several computers manufactures have been building, for many years now, very expensive high-end computers where each computer includes many processors, however only recently relatively cheap multiprocessor computers are available as mainstream computers and can already be found in many homes.

This fundamental change in computing architecture requires a fundamental change in how such computers are programmed. Writing a *concurrent* application for a multiprocessor computer that takes advantage of having multiple processors to increase speed and get better performance, is much more challenging and complex than programming a uniprocessor computer, and requires an understanding of new basic principles.

Synchronization

Many of our daily interactions with other people involve synchronization. You and your spouse may have to synchronize on who will buy the groceries, empty the garbage can, take the kids to school, which one of you will be the first to take a shower (assuming you only have one shower at home), will take the car, or use the single computer you have. Assume that you have a cat and your neighbor has a dog and you and your neighbor are sharing a yard, then you and your neighbor might want to coordinate to make sure that both pets are never in the yard at the same time.

In these examples, synchronization is used to ensure that only one participant (and not both) will take a certain action at a given time. Another type of synchronization has to do with cooperation. You and your spouse might need to move a heavy table together to its new location (it is too heavy for just one person). A classical example of cooperation is for two camps of the same army to decide on the exact time for a coordinated attack on the enemy camp.

All the above examples for synchronization between people have corresponding examples for synchronization between computers. Synchronization is needed in all systems and environments where several processors can be active at the same time. Without proper synchronization, the integrity of the data may be destroyed if two computers (or

[1] Multicore means multiple processors embedded on the same chip.

processors) update a common file at the same time, and as a result, deposits and withdrawals could be lost, confirmed reservations might have disappeared, etc.

While achieving synchronization between humans is relatively easy, achieving synchronization between computers is challenging and difficult. The reason is that computers communicate with each other in a very restricted way. While humans can see and hear each other, computers (or processors) can only read and write. So, one computer can write a note (or send a message) that the other computer will later read, but they can not see each other. To understand the difficulty with this type of restricted communication, we examine in the next section, a simple problem which involves two-person interactions where communication is restricted to writing and reading of notes.

2 Too Much Milk

We use the *too much milk* problem to demonstrate the difficulty involved in synchronization. In this problem the two people involved, let's call them Alice and Bob, can not see each other and they communicate only by writing and reading of notes. In particular, Alice can not see that Bob is reading a note that she has written to him earlier.

Alice and Bob are sharing an apartment. Alice arrives home in the afternoon, looks in the fridge and finds that there is no milk. So, she leaves for the grocery to buy milk. After she leaves, Bob arrives, he also finds that there is no milk and goes to buy milk. At the end they both buy milk and end up with too much milk. So, Alice and Bob are looking for a solution to ensure that:

> *Only one person buys milk, when there is no milk; and someone always buys milk, when there is no milk.*

Notice that a solution in which only Bob is responsible for buying milk is not acceptable. In such a solution, there is a scenario where Alice arrives home and finds that there is no milk, and waits forever for Bob to show up. A proper solution should ensure that, when there is no milk, someone always buy milk even if only one person shows up.

To see the corresponding synchronization problem for computers, replace *milk* in the above example with *file*, and let Alice and Bob be the names of two processors that are trying to avoid updating a shared file at the same time, and they communicate by reading and writing shared memory locations. A solution to the problem should guarantee mutually exclusive access to a *single* shared resource when there are only *two* competing processors. The problem is a special case of the mutual exclusion problem, which is discussed in Section 4.

First Attempt

Alice and Bob have discussed the situation and agreed that in order to synchronize their actions, they will communicate by leaving (signed) notes on the door of the fridge. More specifically, they came up with the following solution:

> *If you find that there is no milk and there is no note on the door of the fridge, then leave a note on the fridge's door, go and buy milk, put the milk in the fridge, and remove your note.*

The problem with this solution is that again both of them might buy milk. To see that, assume that they arrive home at the same time and recall that they can not see each other. Alice finds that there is no milk and that there is no note, so she writes a note and *before* she leaves her note, Bob checks and sees that there is no milk and no note. Thus, both will put their notes and will go to buy milk ending up with "too much milk".

Second Attempt

To resolve the above problem Alice and Bob slightly modified their previous solution.

> *As soon as you arrive home, you leave a note on the fridge's door. Only then you check, and if there is no milk and there is no note (other than yours), then you go and buy milk, put the milk in the fridge and remove your note.*

Well, this time Alice and Bob might end up with no milk at all! To see that, assume that they arrive home at the same time. Each one writes a note and leaves it on the fridge's door. Then, each one finds the note of the other, and no one goes to buy milk.

Third Attempt

Next, we present an "almost" correct solution. It works correctly only if we make a timing assumption about the relative *speed* of Alice and Bob.

> *When Alice arrives home, she leaves a note on the fridge's door. Then, if she finds that there is no milk and that there is no note (signed by Bob), she buys milk, puts the milk in the fridge and removes her note. Otherwise, if Bob left a note, she removes her note, and does nothing (until the day after, when she arrives home and tries again).*
>
> *When Bob arrives home, he leaves a note on the fridge's door. Then if he finds that there is a note signed by Alice, he checks the fridge's door over and over again waiting for Alice to remove her note. Once Bob finds that Alice's note has been removed, he checks if there is milk. If there is no milk, he buys milk, puts the milk in the fridge and removes his note. Otherwise, if there is milk, he removes his note without buying milk.*

Let's assume that Bob is waiting for Alice to remove her note. Then, we should assume that, between the time Alice removes her note, and the time she leaves a new note the day after, Bob must find out that Alice's note has been removed. Without this assumption, Alice and Bob might never buy milk.

A Correct Solution

Finally we present a correct solution. The solution it is symmetric: Alice and Bob behave similarly and hence have the same chance to go and buy milk. For this solution, four labelled notes are used. Alice uses notes *A1* and *A2*, while Bob uses notes *B1* and *B2*.

When Alice arrives home, she does the following:

1. *First, she leaves a note labelled A1 on the fridge's door. Then, if there is a note labelled B2, she leaves a note labelled A2, otherwise she removes A2. By doing so Alice gives priority to Bob in buying milk.*
2. *Then, she checks the fridge's door over and over again as long as the following two conditions hold: (1) there is a note labelled B1, and (2) either both A2 and B2 are present or neither of them is present.*
3. *Once Alice finds that one of these two conditions are not satisfied, she checks if there is milk. If there is no milk, she buys milk, puts it in the fridge, and removes A1. Otherwise, if there is milk, she simply removes A1.*

When Bob arrives home, he does the following:

1. *First, he leaves a note labelled B1 on the fridge's door. Then, if there is no note labelled A2, he leaves a note labelled B2, otherwise he removes B2. By doing so Bob gives priority to Alice in buying milk.*
2. *Then, he checks the fridge's door over and over again as long as the following two conditions hold: (1) there is a note labelled A1, and (2) either A2 or B2 are present (but not both).*
3. *Once Bob finds that one of these two conditions are not satisfied, he checks if there is milk. If there is no milk, he buys milk, puts it in the fridge and removes B1. Otherwise, if there is milk, he simply removes B1.*

The last solution is rather complicated and it is not trivial to formally verify its correctness.[2] The moral from this example is that even for such a simple problem it is challenging to come up with a correct solution when communication is done by reading and writing of notes.

3 Models of Concurrency

A *concurrent* system is a collection of processors that communicate by reading and writing from a shared memory. A *process* corresponds to a given computation. That is, given some program, its execution is 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.

A process may be composed of several *threads*. For example, a process that corresponds to the execution of some game may be composed of three threads: the first controls the keyboard, the second is for doing the computations, and the third is responsible for doing I/O. We will use only the notion of a process; however, all the results apply to both processes and threads.

Interactions between processes or processors – we shall use the terms interchangeably – are of two types: data communication and synchronization. Data communication is the exchange of data, either by sending messages or by reading and writing of shared memory. Synchronization is required when operations of various processes need to obey some order restrictions, and is classified as either cooperation or contention.

[2] For example, Alice and Bob first leave a note, and only then check the fridge's door; the solution would be incorrect if the order is replaced.

Atomic Operations

Most concurrent algorithms assume an architecture in which processes communicate asynchronously via shared objects. Each object has a *type* which defines the set of operations that the object supports. Each object also has *sequential specification* that specifies how the object behaves when these operations are applied sequentially. Asynchrony means that there is no assumptions on the relative speeds of the processes.

Various architecture differ in the level of *atomicity* that is supported. Atomic (or indivisible) operations are defined as operations whose execution is not interfered with by other concurrent activities. 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 atomic 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.
- *Load-link/store-conditional*: the *load-link* operation atomically reads the value of a register r and the *store-conditional* operation tries to atomically write a value into r. The *store-conditional* operation on r by process p succeeds in writing a value, only if no other process has modified r since the last *load-link* operation on r by p. Otherwise, it returns a failure status.
- *Read-modify-write*: In one atomic operation a process can read a value of a shared register, based on the value compute a new value and assign it back to that register.

The definition of atomicity is too restrictive, and it is safe to relax it by assuming that processes can try to access the object at the same time, however, although operations of concurrent processes may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in there "real-time" order. This type of correctness requirement is called *linearizability* [20].

Modern operating systems usually also implement synchronization mechanisms, such as semaphores, that simplify the implementation of locks. 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.

Local vs. Remote Memory Accesses

For certain shared memory systems, it makes sense to distinguish between *local* and *remote* access to shared memory. Consider the following three machine architectures,

1. Central shared memory systems, where a process (or processor) does not has its own private cache, and the shared memory is located in one central location. In such systems, each access to a shared memory location is remote, since it requires to traverse the interconnection network between the process and the memory.
2. Cache coherent systems, where each process has its own private cache. When a process accesses a shared memory location a copy of it migrates to a local cache line and becomes locally accessible until some other process updates this shared memory location and the local copy is invalidated.
3. Distributed shared memory systems, where instead of having the "shared memory" in one central location, each process "owns" part of the shared memory and keeps it in its own local memory. A shared memory location is locally accessible to some process if it is in the part of the shared memory that physically resides on that process local memory.

It is important to try to reduce the number of times a process has to access a shared memory location that is not locally accessible to that process. Notice that spinning (busy-waiting) on a remote memory location while its value does not change, is counted only as *one* remote access that causes communication in a system with support for cache coherence, while it is counted as *many* remote accesses in a central memory system or in a distributed shared memory system.

4 Mutual Exclusion Locks

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.

The problem of constructing a mutual exclusion lock (i.e., the mutual exclusion problem) is the guarantee of mutually exclusive access to a single shared resource when there are several competing processes [9]. The problem arises in operating systems, database systems, multiprocessor computers, and computer networks. It is of great significance, since it lies at the heart of many interprocess synchronization problems.

Definitions

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*.

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.

Key Results

There are hundreds of beautiful algorithms for solving the problem some of which are also very efficient. Because of its importance and as a result of new hardware and software developments, new algorithms are still being designed. Only few representative results 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 [23]. 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 [33].

Lower bounds. A space lower bound when using only atomic registers is that: any deadlock-free mutual exclusion algorithm for n processes must use at least n shared

registers [7]. It was also shown in [7] 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 (i.e., the number of both local and remote memory references) 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) [2].

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 [25], 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 [3, 1, 33]. (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.

Three local-spinning algorithms are presented in [4, 15, 27]. 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 have shown that these algorithms scale very well as contention increases. Local spinning algorithms using only atomic registers are presented in [1, 3, 33].

5 Concurrent Data Structures

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. There are several approaches for the construction of concurrent data structures.

Lock-based Synchronization

Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent data structures: a process accesses the data structure only inside a critical section

code, within which the process is guaranteed exclusive access. The popularity of this approach is largely due to the apparently simple programming model of such locks and the availability of implementations which are efficient and scalable.

When using locks, 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 "small pieces" of a data structure to be locked, 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 mutual exclusion locks to protect the access to a shared data structure may degrade the performance of synchronized concurrent applications, as it forces processes to wait for a lock to be released. Moreover, slow or stopped processes may prevent other processes from ever accessing the data structure. In cases of concurrent updates of simple data structures such as queues, stacks, heaps, linked lists, and counters, locking may be avoided by using *lock-free* data structures.

Lock-free Synchronization

Several progress conditions have been proposed in the literature for lock-free data structures. The two most important conditions are non-blocking [20] and wait-freedom [17].

1. A data structure is *non-blocking* if it guarantees that *some* process will always be able to complete its pending operation in a finite number of its own steps regardless of the execution speed of other processes (admits starvation).
2. A data structure is *wait-free* if it guarantees that *every* process will always be able to complete its pending operations in a finite number of its own steps regardless of the execution speed of other processes (does not admit starvation).

Advantages of using non-blocking algorithms (i.e., algorithms that satisfy the non-blocking property) are that they are not subject to deadlocks or priority inversion, they are resilient to process failures (no data corruption on process failure), and they do not suffer significant performance degradation from scheduling preemption, page faults, or cache misses. Non-blocking algorithms are still not used in many practical applications as such algorithms are often complex (each variant of a non-blocking queue is still a publishable result).

While non-blocking has the potential to significantly improve the performance of concurrent applications, the wait-free property (although desirable) imposes too much overhead upon the implementation. Wait-free algorithms are often very complex and memory consuming, and hence considered less practical than non-blocking algorithms. Furthermore, starvation can be efficiently handled by collision avoidance techniques such as exponential backoff.

The term *lock-free* algorithms refers to algorithms that do not use locking in any way. Lock-free algorithms are designed under the assumption that synchronization conflicts are rare and should be handled only as exceptions; when a synchronization conflict is noticed the operation is simply restarted from the beginning. Non-blocking al-

gorithms are, by definition, also lock-free, but lock-free algorithms are not necessarily non-blocking.[3]

Various other progress conditions weaker than non-blocking have been proposed in the literature. For example, a data structure is *obstruction-free* if it guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" long enough [18]. Efficient lock-free algorithms usually require the use of powerful synchronization primitives such as compare-and-swap or load-linked/store-conditional. Lock-free implementations of various concurrent data structures have appeared in many papers; a few examples are [14, 16, 28, 32, 35].

Transactional Memory

General methodologies for implementing concurrent data structures and algorithms have been proposed in the literature. Such implementations are usually less efficient compared to specialized algorithms. *Transactional memory* is one such methodology which has gained momentum in recent years as a simple way for writing concurrent programs [19, 31]. It is based on the idea of transactions from databases. Transactional memory allows programmers to define customized read-modify-write operations and to apply them to multiple, independently-chosen words of memory as one indivisible step.

The underline implementation of transactional memory may use both locks and lock-free objects, but the complexity is hidden from the programmer. Several lock-based and lock-free implementations have be proposed in the last few years. The transactional memory programming paradigm is also seriously considered by the industry. A list of citations can be found at http://www.cs.wisc.edu/trans-memory/.

6 Multiple Resources

In many situations where synchronization is required, two or more resources are involved. Consider, for example, the problem of transferring money between *two* bank accounts. Here the resources are the two accounts; and the clerks, working in the same bank or in different banks, have to synchronize their actions so that they will not update the same account at the same time.

To transfer money between two bank accounts, a clerk has to (1) lock the two accounts, that is, to get exclusive permission to access the two accounts; (2) transfer the money; (3) release the locks on the two accounts. If two clerks need to transfer money between the same two accounts, and they try to lock the two accounts, one at a time, in different order, then the system may *deadlock*, having each clerk locking one account and waiting for the other.

[3] In the literature, the terms lock-free and non-blocking are sometimes used as synonymous, or even with opposite meaning to the way they are defined in this paper. We find it is useful to distinguish between algorithms that do not require locking (i.e., lock-free algorithms) and those that actually satisfy the non-blocking progress condition.

Deadlocks

How would we "program the clerks" in order to prevent possible deadlocks assuming that there are thousands of clerks and millions of bank accounts, not just two? There is a very simple technique, described in the sequel, which prevents such deadlocks from ever happening. The notion of a deadlock in the context of multiple resources is defined as follows,

> *A set of processes (or processors or computers) is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

The event in the above definition is usually a release of a currently held resource. Thus, in a deadlock situation, there is a set of blocked processes each holding one or more resources and waiting to acquire a resource held by another process in the set. There are various approaches for finding when a system is deadlocked, and then resolving the deadlock situation.[4] However, The best approach for handling deadlocks is to prevent them from ever happening.

Deadlock Prevention

There are several simple design principles which, when applied properly, prevent the occurrence of deadlocks.

The Total Order Technique. The most effective prevention technique is to impose a total ordering of all the resources, and to require that each process requests the resources in an increasing order. More precisely,

> *Assume that all the resources are numbered, and that there is a total order between the numbers of the different resources. If each process requests resources in an increasing order of enumeration, then deadlock is not possible.*

The total order technique guarantees that there is no cycle involving several processes, where each is waiting for a resource held by the next one in the cycle. Thus, deadlock is not possible. This very simple approach, can now be used to solve the problem of transferring money between two bank accounts. The account numbers are used as the resource numbers, and the locking of the two accounts is done in an increasing order. The total order technique guarantees that this solution is indeed without deadlock.

When using the total order technique, if a needed resource is not available, the process waits until the resource becomes available, without releasing any of the resources it has already managed to lock. This type of techniques is called *hold and wait*. In the next two techniques, considered below, if a needed resource is not available, the process may release all the resources it has already managed to lock and starts over. This type of techniques is called *release and wait*.

---

[4] There are various ways to resolve a deadlock: to abort all deadlocked processes; to abort one process at a time until the deadlock is resolved; take a resource from some process; etc.

Two Phase Locking. The two phase locking technique works as follows:

> *Phase one: The process tries to lock all the resources it currently needs, one at a time. If the needed resource is not available, the process releases all the resources it has already managed to lock, and starts over;*
>
> *Phase two: If phase one succeeds, the process uses the resources to perform the work needed and then releases all the resources it had locked in phase one.*

We notice that processes may never finish phase one and hence some kind of a deadlock, called livelock, is possible. This technique was introduced in [12].

Timestamping-ordering. A related method to prevent deadlocks is to select an ordering among the processes and, whenever there is a conflict, let the smaller process win. The most common technique that uses this method is the *timestamping-ordering technique*. Before a process starts to lock the resources it needs, a unique new timestamp is associated with that process. It is required that if a process has been assigned timestamp T_i and later a new process is assigned timestamp T_j then $T_i < T_j$. To implement this method, we associate with each resource a temporary timestamp value, which is the timestamp of the process that is currently holding that resource.

> *Phase one: The process tries to lock all the resources it currently needs, one at a time. If a needed resource is not available and its timestamp value is smaller than that of the process, the process releases all the resources it has already managed to lock, waits until the resource with the smaller timestamp is released, and starts over. Otherwise, if the timestamp value of the resource is not smaller, the process does not release resources, it simply waits until the resource is released and locks it.*
>
> *Phase two: If phase one succeeds, the process uses the resources to perform the work needed and then releases all the resources it had locked in phase one.*

Timestamps can be implemented using the system clock value. Another approach would be to use a logical counter that is incremented every time a new timestamp is assigned. Timestamp-based concurrency-control algorithms are discussed in [5].

Deadlock Avoidance

The deadlock avoidance problem (called by Dijkstra the problem of Deadly Embrace [10]) is to design an algorithm that can be used to decide, by careful allocation of the available resources, whether resources can be efficiently allocated to a requesting processes without running into the danger of reaching a deadlock situation.

7 Classical Synchronization Problems

Several important synchronization problems are briefly described below. Dozens of papers have been published about each one of these problems. Citations and solutions to these problems can be found in [34].

Barrier Synchronization

It is sometimes convenient to design algorithms that are divided into phases such that no process may proceed into the next phase until all processes have finished the current phase and are ready to move into the next phase together. In such algorithms, each phase typically depends on the results of the previous phase. This type of behavior can be achieved by placing a *barrier* at the end of each phase.

A barrier is a coordination mechanism (an algorithm) that forces processes which participate in a concurrent algorithm to wait until each one of them has reached a certain point in its program. The collection of these coordination points is called the barrier. Once all the processes have reached the barrier, they are all permitted to continue past the barrier.

Using barriers, a programmer may design a concurrent algorithm under the assumption that the algorithm should work correctly only when it executes in a *synchronous* environment (where the processes run at the same speed or where the processes share a global clock). Then, by using barriers for synchronization, the algorithm can be adapted to work also in an *asynchronous* environment. Such an approach is particularly helpful for the design of numerical and scientific algorithms [21].

The ℓ-exclusion Problem

The ℓ-*exclusion* problem, a natural generalization of the mutual exclusion problem, is to design an algorithm which guarantees that up to ℓ processes and no more may simultaneously access identical copies of the same non-sharable resource when there are several competing processes. That is, ℓ processes are permitted to be in their critical sections simultaneously. A solution is required to withstand the slow-down or even the crash (fail by stopping) of up to $\ell-1$ processes [13].

To illustrate the problem, consider a bank where people are waiting for a teller. Here the processes are the people, the resources are the tellers, and the parameter ℓ is the number of tellers. We notice that the usual bank solution, where people line up in a single queue, and the person at the head of the queue goes to any free teller, does *not* solve the ℓ-exclusion problem. If $\ell \geq 2$ tellers are free, a proper solution should enable the first ℓ people in line to move simultaneously to a teller. However, the bank solution, requires them to move past the head of the queue one at a time. Moreover, if the person at the front of the line "fails", then the people behind this person wait forever. Thus, a better solution is required which will not allow a single failure to tie up all the resources.

The Producer-Consumer Problem

Assume that there are two groups of processes the producers and the consumers and that communication between these two groups is achieved by using a shared buffer. The producers add data items to the end of the buffer while the consumers remove (consume) data items from the beginning of the buffer.

The producer-consumer problem is to write the code for the producers that is used to access the buffer in order to add new produced data items, and the code for the consumers that is used to access the buffer in order to consume data items. The code

must satisfy the following requirements: every data item that is produced is eventually consumed; producers wait for consumers only when the buffer is full; consumers wait for producers only when the buffer is empty; and deadlock is not possible [10].

Two versions of the problem exist: the unbounded-buffer version, which places no limit on the size of the buffer; and the bounded-buffer version, which assumes that there is a fixed buffer size.

Readers and Writers

Consider the example of an *airline reservation system* where multiple processes can *read* the database at the same time, but if one process is *writing*, no other process is allowed to access the database. One simple way to implement such a system is to use a mutual exclusion lock and to require that an access to the database is done only while in a critical section. Such a solution would be very inefficient as it prevents readers from accessing the database at the same time.

This issue is addressed by the readers and writers problem [8]. In this problem, it is assumed that there are two groups of processes, the *readers* and the *writers*. As in the mutual exclusion problem a process starts by executing its 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 the following (safety) condition always holds: There is no limit on the number of readers that can be in their critical sections simultaneously, that is, the readers may share the critical section with each other. However, the writers must have exclusive access, that is, a writer can be in its critical section only when no other process (a reader or another writer) is in its critical section.

Once a process finishes its critical section, it executes its exit code in which it notifies other processes that it is no longer in its critical section, and returns to the remainder.

More Synchronization Problems

Many more synchronization problems were defined over the last 40 years. Few of these problems are: group mutual exclusion [22] (also called room synchronization [6]); choice coordination [30]; concurrent reading and writing [24] (also called concurrent reading while writing); concurrent reading and writing of clocks [26]; the dining philosophers problem [11]; the sleeping barber [10, 11]; and the cigarette smoker's problem [29].

8 The Relative Power of Shared Objects

Are there problems that can be solved using one type of shared objects and can not be solved using another type? We are addressing the following question: Given two objects o_1 and o_2, is there a wait-free implementation[5] of a single object of type o_1 from objects of type o_2 and atomic registers? As is shown in [17], the answer depends on whether the consensus problem, defined below, can be solved using these two objects.

[5] A *wait-free* implementation guarantees that any process can complete any operation in a finite number of steps, regardless of the speed of the other processes.

Consensus

The consensus problem is to design an algorithm in which all correct processes reach a common decision based on their initial opinions. The problem is defined as follows. There are n processes $p_1, p_2, \ldots, p_n$. Each process p_i has an input value $x_i \in \{0, 1\}$. The requirements of the consensus problem are that there exists a *decision value* v such that: (1) each non-faulty process eventually decides on v, and (2) $v \in \{x_1, x_2, \ldots, x_n\}$.

In particular, if all input values are the same, then that value must be the decision value. The problem is a fundamental coordination problem and is at the core of many algorithms for fault-tolerant applications. Achieving consensus is straightforward in the absence of faults, for all processes can share their initial values and then all apply a uniform decision rule to the resulting vector of initial values. However, such a simple algorithm cannot tolerate even a single process crash, for its correctness depends on all processes obtaining the same initial value vector.

Consensus Numbers and the Consensus Hierarchy

We define two notions for measuring the computational power of shared objects.

- The *consensus number* of an object of type o, is the largest n for which it is possible to solve consensus for n processes in a wait-free manner, using any number of objects of type o and any number of atomic registers. If no largest n exists, the consensus number of o is infinite.
- The *consensus hierarchy* (also called the wait-free hierarchy) is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with consensus number i.

Next, we state an important result in the area of concurrent computing [17].

> *In the consensus hierarchy, for any positive i, in a system with i processes: (1) no object at level less than i together with atomic registers can implement any object at level i; and (2) each object at level i together with atomic registers can implement any object at level i or at a lower level.*

Classifying objects by their consensus numbers is a powerful technique for understanding the relative power of shared objects. It can help in deciding which synchronization primitives multi-processor architectures should support. The consensus numbers of several interesting objects are summarized in the following table.

Consensus Number	Object
1	atomic-register, atomic-snapshot, safe^m, regular^m
2	test-and-set, fetch-and-increment, fetch-and-add, swap, queue, stack, read-modify-write bit, fetch-and-increment^m, fetch-and-add^m
$\Theta(\sqrt{m})$	swap^m
$2m-2$	m-register assignment, atomic-register^m $(m > 1)$
∞	compare-and-swap, LL/SC, sticky-bit, queue^2 3-valued read-modify-write, augmented-queue

In the table we refer to *multi-objects* in which a process may simultaneously access several objects in one atomic operation. That is, given a system with a set of component objects of type o, and a parameter m, the multi-object o^m is an object in which a process is allowed to simultaneously (and atomically) execute operations on up to m of the component objects of type o. Examples are a $register^m$ multi-object which allows processes to read and write up to m registers in a single atomic operation; and a $queue^2$ multi-object which allows processes to enqueue and dequeue up to two queues in a single atomic operation.

The $register^m$ object generalizes the *m-register assignment*, which supports writes to m registers in a single atomic operation, and the *atomic-snapshot* object, which supports reads of multiple registers in a single atomic operation. An *augmented queue* is a queue that in addition to the usual enqueue and dequeue operations, also supports a *peek* operation which returns the oldest value in the queue but does not remove it.

The Universality of Consensus

An object o is *universal* if any object which has sequential specification has a wait-free linearizable implementation using atomic registers and objects of type o. In [17], it is proved that consensus for n processes is universal in a system with n processes, for any positive n. An immediate implication of this result is that, an object o is universal in a system with n processes if and only if the consensus number of o is at least n. Thus, in a system of n processes, objects with consensus number at least n, such as compare-and-swap and LL/SC (load-link/store-conditional), are universal objects.

References

1. Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 30:67–86, 2002.
2. R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proc. of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, 1992.
3. J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proc. of the 14th international symposium on distributed computing. LNCS* 1914:29–43, 2000.
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, 1990.
5. P.A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. of the International Conference on Very Large Databases*, pages 285–300, 1980.
6. G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronization. In *Proc. of the 13th Annual Symposium on Parallel Algorithms and Architectures*, pages 122–133, 2001.
7. J.N. Burns and N.A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.
8. P.L. Courtois, F. Heyman, and D.L Parnas. Concurrent control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, 1971.
9. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
10. 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.

11. E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
12. K.P. Eswaran, J.N. Gary, A. Lorie, and I.L Traiger. The notion of consistency and predicate locks in database systems. *Communications of the ACM*, 19(11):624–633, 1976.
13. M.J. Fischer, N. A.Lynch, J.E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM TOPLAS*, 11(1):90–114, 1989.
14. M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 50–59, 2004.
15. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computers*, 28(6):69–69, 1990.
16. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th international symp. on distributed computing*, *LNCS* 2180:300–314, 2003.
17. M. P. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991.
18. M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.
19. M. P. Herlihy and J.E.B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.
20. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.
21. H. Jordan. A special purpose architecture for finite element analysis. In *Proc. of the Int. Conf. on Parallel Processing*, pages 263–266, 1978.
22. Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.
23. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
24. L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.
25. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.
26. L. Lamport. Concurrent reading and writing of clocks. *ACM Trans. on Computer Systems*, 8(4):305–310, 1990.
27. 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.
28. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.
29. S. S. Patil. Limitations of capabilities of Dijkstra's semaphore primitives for coordination among processes. In *Project MAC computational structures group, MIT, memo 57*, 1971.
30. M. O. Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.
31. N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, 1995.
32. H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *8th International Conference on Principles of Distributed Systems*, 2004.
33. G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, 2004. *LNCS* 3274:56–70, 2004.
34. G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education - Prentice-Hall, 2006. ISBN: 0131972596. www.faculty.idc.ac.il/gadi/book.htm.
35. J. D. Valois. Implementing lock-free queues. In *Proc. of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 212–222, 1994.