SPRINGER
REFERENCE

David Padua
Editor-in-Chief

Encyclopedia of Parallel Computing

Springer

David Padua (Ed.)

Encyclopedia of Parallel Computing

With 880 Figures and 98 Tables

Springer

Editor-in-Chief
David Padua
University of Illinois at Urbana-Champaign
Urbana, IL
USA

ISBN 978-0-387-09765-7 e-ISBN 978-0-387-09766-4
DOI 10.1007/978-0-387-09766-4
Print and electronic bundle ISBN: 978-0-387-09844-9
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011935063

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Synchronization

Michael L. Scott
University of Rochester, Rochester, NY, USA

Synonyms

Fences; Multiprocessor synchronization; Mutual exclusion; Process synchronization

Definition

Synchronization is the use of language or library mechanisms to constrain the ordering (interleaving) of instructions performed by separate threads, to preclude orderings that lead to incorrect or undesired results.

Discussion

In a parallel program, the instructions of any given thread appear to occur in sequential order (at least from that thread's point of view), but if the threads run independently, their sequences of instructions may interleave arbitrarily, and many of the possible interleavings may produce incorrect results. As a trivial example, consider a global counter incremented by multiple threads. Each thread loads the counter into a register, increments the register, and writes the updated value back to memory. If two threads load the same value before either stores it back, updates may be lost:

```
            c == 0
Thread 1:                Thread 2:
    r1 := c
                                r1:= c
    ++r1
                                ++r1
    c := r1
                                c := r1
            c == 1
```

Synchronization serves to preclude invalid thread interleavings. It is commonly divided into the subtasks of *atomicity* and *condition synchronization*. Atomicity ensures that a given sequence of instructions, typically performed by a single thread, appears to all other threads as if it had executed indivisibly – not interleaved with anything else. In the example above, one would typically specify that the load-increment-store instruction sequence should execute atomically.

Condition synchronization forces a thread to wait, before performing an operation on shared data, until some desired precondition is true. In the example above, one might want to wait until all threads had performed their increments before reading the final count.

While it is tempting to suspect that condition synchronization subsumes atomicity (make the precondition be that no other thread is currently executing a conflicting operation), atomicity is in fact considerably harder, because it requires *consensus* among all competing threads: they must all agree as to which will proceed and which will wait. Put another way, condition synchronization delays a thread until some locally observable condition is seen to be true; atomicity is a property of the system as a whole.

Like many aspects of parallel computing, synchronization looks different in shared-memory and message-passing systems. In the latter, synchronization is generally subsumed in the message-passing methods; in a shared-memory system, it typically employs a separate set of methods.

Shared-memory implementations of synchronization can be categorized as *busy-wait* (*spinning*), or *scheduler-based*. The former actively consume processor cycles until the running thread is able to proceed. The latter deschedule the current thread, allowing the processor to be used by other threads, with the expectation that future activity by one of those threads will make the original thread runnable again. Because it avoids the cost of two context switches, busy-wait synchronization is typically faster than scheduler-based synchronization when the expected wait time is short and when the processor is not needed for other purposes. Scheduler-based synchronization is typically faster when expected wait times are long; it is *necessary* when the number of threads exceeds the number of processors (else quantum-long delays or even deadlock can occur). In the typical implementation, busy-wait synchronization is built on top of whatever hardware instructions execute atomically. Scheduler-based synchronization, in turn, is built on top of busy-wait synchronization, which is used to protect the scheduler's own data structures (see entries on *Scheduling Algorithms* and on *Processes, Tasks, and Threads*).

Hardware Primitives

In the earliest multiprocessors, *load* and *store* were the only memory-access instructions guaranteed to be atomic, and busy-wait synchronization was implemented using these. Modern machines provide a variety of atomic *read-modify-write* (RMW) instructions, which serve to *update* a memory location atomically. These significantly simplify the implementation of synchronization. Common RMW instructions include:

Test-and-set (l) sets the Boolean variable at location l to *true*, and returns the previous value.

Swap (l, v) stores the value v to location l and returns the previous value.

Atomic-ϕ (l, v) replaces the value o at location l with $\phi(o, v)$ for some simple arithmetic function ϕ (add, sub, and, etc.).

Fetch-and-ϕ (l, v) is like atomic-ϕ, but also returns the previous value.

Compare-and-swap (l, o, n) inspects the value v at location l, and if it is equal to o, replaces it with n. In either case, it returns the previous value, from which one can deduce whether the replacement occurred.

Load-linked (l) and store-conditional (l, v). The first of these returns the value at location l and "remembers" l. The second stores v to l if l has not been modified by any other processor since a previous load-linked by the current processor.

These instructions differ in their expressive power. Herlihy has shown [9] that compare-and-swap (CAS) and load-linked / store-conditional (LL/SC) are *universal* primitives, meaning, informally, that they can be used to construct a *non-blocking* implementation of any other RMW operation. The following code provides a simple implementation of fetch-and-ϕ using CAS.

```
val old := *l;
loop
    val new := phi(old);
    val found := CAS(l, old, new);
    if (old == found) break;
    old := found;
```

If the test on line 5 of this code fails, it must be because some other thread successfully modified `*l`. The system as a whole has made forward progress, but the current thread must try again.

As discussed in the entry on Non-blocking Algorithms, this simple implementation is *lock-free* but not *wait-free*. There are stronger (but slower and more complex) non-blocking implementations in which each thread is guaranteed to make forward progress in a bounded number of its own instructions.

NB: In any distributed system, and in most modern shared memory systems, instructions executed by a given thread are not, in general, guaranteed to be seen in sequential order by other threads, and instructions of any two threads are not, in general, guaranteed to be seen in the same order by all of their peers. Modern processors typically provide so-called *fence* or *barrier* instructions (not to be confused with the barriers discussed under Condition Synchronization below) that force previous instructions of the current thread to be seen by other threads before subsequent instructions of the current thread. Implementations of synchronization

methods typically include sufficient fences that if synchronization method s_1 in thread t_1 occurs before synchronization method s_2 in thread t_2, then all instructions that precede s_1 in t_1 will appear in t_2 to have occurred before any of its own instructions that follow s_2. For more information, see the entry on Memory Models. The remainder of the discussion here assumes that memory is *sequentially consistent*, that is, that instructions appear to interleave in some global total order that is consistent with program order in every thread.

Atomicity

A multi-instruction operation is said to be *atomic* if appears to occur "all at once" from every other thread's point of view. In a sequentially consistent system, this means that the program behaves as if the instructions of the atomic operation were contiguous in the global instruction interleaving. More specifically, in any system, intermediate states of the atomic operation should never be visible to other threads, and actions of other threads should never become visible to a given thread in the middle of one of its own atomic operations.

The most straightforward way to implement atomicity is with a *mutual-exclusion (mutex) lock* – an abstract object that can be *held* by at most one thread at a time. In standard usage, a thread invokes the *acquire* method of the lock when it wishes to begin an atomic operation and the *release* method when it is done. *Acquire* waits (by spinning or rescheduling) until it is safe for the operation to proceed. The code between the acquire and release (the body of the atomic operation) is known as a *critical section*.

Critical sections that conflict with one another (typically, that access some common location, with at least one section writing that location) must be protected by the same lock. Programming discipline commonly ensures this property by associating data with locks. A thread must then acquire locks for all the data accessed in a critical section. It may do so all at once, at the beginning of the critical section, or it may do so incrementally, as the need for data is encountered. Considerable care may be required to ensure that locks are acquired in the same order by all critical sections, to avoid deadlock. All locks are typically held until the end of the critical section. This *two-phase locking* (all acquires occur before any releases) ensures that the global set of critical section executions remains *serializable*.

Relaxations of Mutual Exclusion

So-called *reader–writer locks* increase concurrency by observing that it is safe for more than one thread to read a location concurrently, so long as no thread is modifying that location. Each critical section is classified as either a reader or a writer of the data associated with a given lock. The *reader_acquire* method waits until there is no concurrent writer of the lock; the *writer_acquire* method waits until there is no concurrent reader *or* writer.

In a standard reader–writer lock, a thread must know, when it first reads a location, whether it will ever need to write that location in the current critical section. In some contexts it may be possible to relax this restriction. The Linux kernel, for example, provides a *sequence lock* mechanism that allows a reader to *abort* its peers and upgrade to writer status. Programmers are required to follow a restrictive programming discipline that makes critical sections "restartable," and checks, before any write or "dangerous" read, to see whether a peer's upgrade has necessitated a restart.

For data structures that are almost always read, and very occasionally written, several operating system kernels provide some variant of a mechanism known as *RCU* (originally an abbreviation for read-copy update). RCU divides execution into so-called *epochs*. A writer creates a new copy of any data structure it needs to update. It replaces the old copy with the new, typically using a single CAS instruction. It then waits until the end of the current epoch to be sure that all readers that might have been using the old copy have completed their critical sections (at which point it can reclaim the old copy, or perform other actions that depend on the visibility of the update). The advantage of RCU, in comparison to locks, is that it imposes *zero overhead* in the read-only case.

For more general-purpose use, *transactional memory* (TM) allows arbitrary operations to be executed atomically, with an underlying implementation based on *speculation* and *rollback*. Originally proposed [10] as a hardware assist for lock-free data structures – sort of a multi-word generalization of LL/SC – TM has seen a flurry of activity in recent years, and several hardware

and software implementations are now widely available. Each keeps track of the memory locations accessed by transactions (would-be atomic operations). When two concurrent transactions are seen to conflict, at most one is allowed to *commit*; the others *abort*, "roll back," and try again, using a fully automated, transparent analogue of the programming discipline required by sequence locks. For further details, see the separate entry on TM.

Fairness

Because they sometimes force multiple threads to wait, synchronization mechanisms inevitably raise issues of *fairness*. When a lock is released by the current holder, which waiting thread should be allowed to acquire it? In a system with reader–writer locks, should a thread be allowed to join a group of already-active readers when writers are already waiting? When transactions conflict in a TM system, which should be permitted to proceed, and which should wait or abort?

Many answers are possible. The choice among conflicting threads may be arbitrary, random, first-come-first-served (FIFO), or based on some other notion of priority. From the point of view of an individual thread, the resulting behavior may range from potential *starvation* (no progress guarantees) to some sort of proportional share of system run time. Between these extremes, a thread may be guaranteed to run eventually if it is continuously ready, or if it is ready infinitely often. Even given the possibility of starvation, the system as a whole may be *livelock-free* (guaranteed to make forward progress) as a result of algorithmic guarantees or pseudo-random heuristics. (Actual livelock is generally considered unacceptable.) Any starvation-free system is clearly livelock free.

Simple Busy-Wait Locks

Several early locking algorithms were based on only loads and stores, but these are mainly of historical interest today. All required $\Omega(tn)$ space for t threads and n locks, and $\omega(1)$ (more-than-constant) time to arbitrate among threads competing for a given lock.

In modern usage, the simplest constant-space, busy-wait mutual exclusion lock is the *test-and-set (TAS) lock*, in which a thread acquires the lock by using a test-and-set instruction to change a Boolean flag from false to true. Unfortunately, spinning by waiting threads tends to induce extreme contention for the lock location, tying up bus and memory resources needed for productive work. On a cache-coherent machine, better performance can be achieved with a "test-and-test-and-set" (TATAS) lock, which reduces contention by using ordinary load instructions to spin on a value in the local cache so long as the lock remains held:

```
type lock = Boolean;

proc acquire(lock *l):
     while (test-and-set(l))
        while (*l) /* spin */ ;

proc release(lock *l):
    *l := false;
```

This lock works well on small machines (up to, say, four processors).

Which waiting thread acquires a TATAS lock at release time depends on vagaries of the hardware, and is essentially arbitrary. Strict FIFO ordering can be achieved with a *ticket lock*, which uses fetch-and-increment (FAI) and a pair of counters for constant space and (per-thread) time. To acquire the lock, a thread atomically performs an FAI on the "next available" counter and waits for the "now serving" counter to equal the value returned. To release the lock, a thread increments its own ticket, and stores the result to the "now serving" counter. While arguably fairer than a TATAS lock, the ticket lock is more prone to performance anomalies on a multiprogrammed system: if any waiting thread is preempted, all threads behind it in line will be delayed until it is scheduled back in.

Scalable Busy-Wait Locks

On a machine with more than a handful of processors, TATAS and ticket locks scale poorly, with time per critical section growing linearly with the number of waiting threads. Anderson [1] showed that exponential backoff (reminiscent of the Ethernet contention-control algorithm) could substantially improve the performance of TATAS locks. Mellor-Crummey and Scott [17] showed similar results for linear backoff in ticket locks (where

a thread can easily deduce its distance from the head of the line).

To eliminate contention entirely, waiting threads can be linked into an explicit queue, with each thread spinning on a separate location that will be modified when the thread ahead of it in line completes its critical section. Mellor-Crummey and Scott showed how to implement such queues in total space $O(t + n)$ for t threads and n locks; their *MCS lock* is widely used in large-scale systems. Craig [4] and, independently, Landin and Hagersten [16] developed an alternative *CLH lock* that links the queue in the opposite direction and performs slightly faster on some cache-coherent machines. Auslander et al. developed a variant of the MCS lock that is API-compatible with traditional TATAS locks [3]. Kontothanassis et al. [14] and He et al. [8] developed variants of the MCS and CLH locks that avoid performance anomalies due to preemption of threads waiting in line.

Scheduler-Based Locks

A busy-wait lock wastes processor resources when expected wait times are long. It may also cause performance anomalies or deadlock in a multiprogrammed system. The simplest solution is to *yield* the processor in the body of the spin loop, effectively moving the current thread to the end of the scheduler's ready list and allowing other threads to run. More commonly, *scheduler-based locks* are designed to *deschedule* the waiting thread, moving it (atomically) from the ready list to a separate queue associated with the lock. The release method then moves one waiting thread from the lock queue to the ready list. To minimize overhead when waiting times *are* short, implementations of scheduler-based synchronization commonly spin for a small, bounded amount of time before invoking the scheduler and yielding the processor. This strategy is often known as *spin-then-wait*.

Condition Synchronization

It is tempting to assume that busy-wait condition synchronization can be implemented trivially with a Boolean flag: a waiting thread spins until the flag is true; a thread that satisfies the condition sets the flag to true. On most modern machines, however, additional fence instructions are required both in the satisfying thread, to ensure that its prior writes are visible to other threads, and in the waiting thread, to ensure that its subsequent reads do not occur until after the spin completes. And even on a sequentially consistent machine, special steps are required to ensure that the compiler does not violate the programmer's expectations by reordering instructions within threads.

In some programming languages and systems, a variable may be made suitable for condition synchronization by labeling it `volatile` (or, in C++'0X, `atomic<>`). The compiler will insert appropriate fences at reads and writes of `volatile` variables, and will refrain from reordering them with respect to other instructions.

Some other systems provide special *event* objects, with methods to set and await them. Semaphores and monitors, described in the following two subsections, can be used for both mutual exclusion and condition synchronization.

In systems with dynamically varying concurrency, the *fork* and *join* methods used to create threads and to verify their completion can be considered a form of condition synchronization. (These are, in fact, the principal form of synchronization in systems like Cilk and OpenMP.)

Barriers

One form of condition synchronization is particularly common in data-parallel applications, where threads iterate together through a potentially large number of algorithmic phases. A *synchronization barrier*, used to separate phases, guarantees that no thread continues to phase $n + 1$ until all threads have finished phase n.

In most (though not all) implementations, the barrier provides a single method, composed internally of an *arrival* phase that counts the number of threads that have reached the barrier (typically via a log-depth tree) and a *departure* phase in which permission to continue is broadcast back to all threads. In a so-called *fuzzy barrier* [6], these arrival and departure phases may be separate methods. In between, a thread may perform any instructions that neither depend on the arrival of other threads nor are required by other threads prior to their departure. Such instructions can serve to "smooth out" phase-by-phase imbalances in the work assigned

to different threads, thereby reducing overall wait time. Wait time may also be reduced by an *adaptive barrier* [7, 19], which completes the arrival phase in constant time after the arrival of the final thread.

Unfortunately, when t threads arrive more or less simultaneously, no barrier implementation using ordinary loads, stores, and RMW instructions can complete the arrival phase in less than $\Omega(\log t)$ time. Given the importance of barriers in scientific applications, some supercomputers have provided special near-constant-time hardware barriers. In some cases the same hardware has supported a fast *eureka* method, in which one thread can announce an event to all others in constant time.

Semaphores

First proposed by Dijkstra in 1965 [5] and still widely used today, *semaphores* support both mutual exclusion and condition synchronization. A *general semaphore* is a nonnegative counter with an initial value and two methods, known as **V** and **P**. The **V** method increases the value of the semaphore by one. The **P** method waits for the value to be positive and then decreases it by one. A *binary semaphore* has values restricted to zero and one (it is customarily initialized to one), and serves as a mutual exclusion lock. The **P** method acquires the lock; the **V** method releases the lock. Programming discipline is required to ensure that **P** and **V** methods occur in matching pairs.

The typical implementation of semaphores pairs the counter with a queue of waiting threads. The **V** method checks to see whether the counter is currently zero. If so, it checks to see whether any threads are waiting in the queue and, if there are, moves one of them to the ready list. If the counter is already positive (in which case the queue is guaranteed to be empty) or if the counter is zero but the queue is empty, **V** simply increments the counter. The **P** method also checks to see whether the counter is zero. If so, it places the current thread on the queue and calls the scheduler to yield the processor. Otherwise it decrements the counter.

General semaphores can be used to represent resources of which there is a limited number, but more than one. Examples include I/O devices, communication channels, or free or full slots in a fixed-length buffer. Most operating systems provide semaphores as part of the kernel API.

Monitors

While semaphores remain the most widely used scheduler-based shared-memory synchronization mechanism, they suffer from several limitations. In particular, the association between a binary semaphore (mutex lock) and the data it protects is solely a matter of convention, as is the paired usage of **P** and **V** methods. Early experience with semaphores, combined with the development of language-level abstraction mechanisms in the 1970s, led several developers to suggest building higher-level synchronization abstractions into programming languages. These efforts culminated in the definition of *monitors* [12], variants of which appear in many languages and systems.

A monitor is a data abstraction (a module or class) with an implicit mutex lock and an optional set of *condition variables*. Each *entry* (method) of the monitor automatically acquires and releases the mutex lock; entry invocations thus exclude one another in time. Programmers typically devise, for each monitor, a program-specific *invariant* that captures the mutual consistency of the monitor's state (data members – fields). The invariant is assumed to be true at the beginning of each entry invocation, and must be true again at the end.

Condition variables support a pair of methods superficially analogous to **P** and **V**; in Hoare's original formulation, these were known as *wait* and *signal*. Unlike **P** and **V**, these methods are *memory-less*: a signal invocation is a no-op if no thread is currently waiting.

For each reason that a thread might need to wait within a monitor, the programmer declares a separate condition variable. When it waits on a condition, the thread releases exclusion on the monitor. The programmer must thus ensure that the invariant is true immediately prior to every wait invocation.

Semantic Details

The details of monitors vary significantly from one language to another. The most significant issues, discussed in the paragraphs below, are commonly known as the *nested monitor problem* and the modeling of signals as *hints vs. absolutes*. More minor issues include language syntax, alternative names for signal and wait, the modeling of condition variables in the type system, and the prioritization of threads waiting for conditions or for access to the mutex lock.

The nested monitor problem arises when an entry of one monitor invokes an entry of another monitor, and the second entry waits on a condition variable. Should the wait method release exclusion on the outer monitor? If it does, there is no guarantee that the outer monitor will be available again when execution is ready to resume in the inner call. If it does not, the programmer must take care to ensure that the thread that will perform the matching signal invocation does not need to go through the outer monitor in order to reach the inner one. A variety of solutions to this problem have been proposed; the most common is to leave the outer monitor locked.

Signal methods in Hoare's original formulation were defined to transfer monitor exclusion directly from the signaler to the waiter, with no intervening execution. The purpose of this convention was to guarantee that the condition represented by the signal was still true when the waiter resumed. Unfortunately, the convention often has the side effect of inducing extra context switches, and requires that the monitor invariant be true immediately prior to every signal invocation. Most modern monitor variants follow the lead of Mesa [15] in declaring that a signal is merely a hint, and that a waiting process must double-check the condition before continuing execution. In effect, code that would be written

```
if (!condition)
    cond_var.wait();
```

in a Hoare monitor is written

```
while (!condition)
    cond_var.wait();
```

in a Mesa monitor. To make it easier to write programs in which a condition variable "covers" a set of possible conditions (particularly when signals are hints), many monitor variants provide a *signal-all* or *broadcast* method that awakens all threads waiting on a condition, rather than only one.

Message Passing

In a system in which threads interact by exchanging messages, rather than by sharing variables, synchronization is generally implicit in the *send* and *receive* methods. A receive method typically blocks until an appropriate message is available (a matching send has been performed). Blocking semantics for send methods vary from one system to another:

Asynchronous send – In some systems, a sender continues execution immediately after invoking a send method, and the underlying system takes responsibility for delivering the message. While often desirable, this behavior complicates the delivery of failure notifications, and may be limited by finite buffering capacity.

Synchronous send – In other systems – notably those based on Hoare's Communicating Sequential Processes (CSP) [13] – a sender waits until its message has been received.

Remote-invocation send – In yet other systems, a send method has both ingoing and outcoming parameters; the sender waits until a reply is received from its peer.

Distributed Locking

Libraries, languages, and applications commonly implement higher-level distributed locks or transactions on top of message passing. The most common lock implementation is analogous to the MCS lock: acquired requests are sent to a *lock manager* thread. If the lock is available, the manager responds directly; otherwise it forwards the request to the last thread currently waiting in line. The release method sends a message to the manager or, if a forwarding request has already been received, to the next thread in line for the lock. Races in which the manager forwards a request at the same time the last lock holder sends it a release are trivially resolved by statically choosing one of the two (perhaps the lock holder) to inform the next thread in line. Distributed transaction systems are substantially more complex.

Rendezvous and Remote Procedure Call

In some systems, a message must be received explicitly by an already existing thread. In other systems, a thread is created by the underlying system to handle each arriving message. Either of these options – *explicit* or *implicit receipt* – can be paired with any of the three send options described above. The combination of remote-invocation send with implicit receipt is often called *remote procedure call* (RPC). The combination of remote-invocation send with explicit receipt is known

as *rendezvous*. Interestingly, if all shared data is encapsulated in monitors, one can model – or implement – each monitor with a *manager* thread that executes entry calls one at a time. Each such call then constitutes a rendezvous between the sender and the monitor.

Related Entries

►Actors
►Cache Coherence
►Concurrent Collections Programming Model
►Deadlocks
►Memory Models
►Monitors, Axiomatic Verification of
►Non-Blocking Algorithms
►Path Expressions
►Processes, Tasks, and Threads
►Race Conditions
►Scheduling Algorithms
►Shared-Memory Multiprocessors
►Transactions, Nested

Bibliographic Notes

The study of synchronization began in earnest with Dijkstra's "Cooperating Sequential Processes" monograph of 1965 [5]. Andrews and Schneider provide an excellent survey of synchronization mechanisms circa 1983 [2]. Mellor-Crummey and Scott describe and compare a variety of busy-wait spin locks and barriers, and introduce the MCS lock [17]. More extensive coverage of synchronization can be found in Chapter 12 of Scott's programming languages text [18], or in the recent texts of Herlihy and Shavit [11] and Taubenfeld [20].

Bibliography

1. Anderson TE (Jan 1990) The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans Parallel Distr Sys 1(1):6–16
2. Andrews GR, Schneider FB (Mar 1983) Concepts and notations for concurrent programming. ACM Comput Surv 15(1):3–43
3. Auslander MA, Edelsohn DJ, Krieger OY, Rosenburg BS, Wisniewski RW (2003) Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application 20030200457, submitted 23 Oct 2003
4. Craig TS (Feb 1993) Building FIFO and priority-queueing spin locks from atomic swap. Technical Report 93-02-02, University of Washington Computer Science Department
5. Dijkstra EW (Sept 1965) Cooperating sequential processes. Technical report, Technological University, Eindhoven, The Netherlands. Reprinted in Genuys F (ed) Programming Languages, Academic Press, New York, 1968, pp 43–112. Also available at www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html.
6. Gupta R (Apr 1989) The fuzzy barrier: a mechanism for high speed synchronization of processors. Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, pp 54–63
7. Gupta R, Hill CR (June 1989) A scalable implementation of barrier synchronization using an adaptive combining tree. Int J Parallel Progr 18(3):161–180
8. He B, Scherer III WN, Scott ML (Dec 2005) Preemption adaptivity in time-published queuebased spin locks. Proceeding of the 2005 International Conference on High Performance Computing, Goa, India
9. Herlihy MP (Jan 1991) Wait-free synchronization. ACM Trans Progr Lang Syst 13(1):124–149
10. Herlihy MP, Moss JEB (1993) Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th International Symposium on Computer Architecture, San Diego, CA, May 1993 pp 289–300
11. Herlihy MP, Shavit N (2008) The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington, MA
12. Hoare CAR (Oct 1974) Monitors: an operating system structuring concept. Commun ACM 17(10):549–557
13. Hoare CAR (Aug 1978) Communicating sequential processes. Commun ACM 21(8):666–677
14. Kontothanassis LI, Wisniewski R, Scott ML (Feb 1997) Scheduler-conscious synchronization. ACM Trans Comput Sys 15(1):3–40
15. Lampson BW, Redell DD (Feb 1980) Experience with processes and monitors in Mesa. Commun ACM 23(2):105–117
16. Magnussen P, Landin A, Hagersten E (Apr 1994) Queue locks on cache coherent multiprocessors. Proceedings of the 8th International Parallel Processing Symposium, Cancun, Mexico, pp 165–171
17. Mellor-Crummey JM, Scott ML (Feb 1991) Algorithms for scalable synchronization on sharedmemory multiprocessors. ACM Trans Comput Syst 9(1):21–65
18. Scott ML (2009) Programming Language Pragmatics, 3rd edn. Morgan Kaufmann, Burlington, MA
19. Scott ML, Mellor-Crummey JM (Aug 1994) Fast, contention-free combining tree barriers. Int J Parallel Progr 22(4):449–481
20. Taubenfeld G (2006) Synchronization Algorithms and Concurrent Programming. Prentice Hall, Upper Saddle River