Shared-Memory Synchronization

MIchael L. Scott
University of Rochester

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #1

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Ever since the advent of time sharing in the 1960s, designers of concurrent and parallel systems have needed to synchronize the activities of threads of control that share data structures in memory. In recent years, the study of synchronization has gained new urgency with the proliferation of multicore processors, on which even relatively simple user-level programs must frequently run in parallel.

This monograph offers a comprehensive survey of shared-memory synchronization, with an emphasis on "systems-level" issues. It includes sufficient coverage of architectural details to understand correctness and performance on modern multicore machines, and sufficient coverage of higher-level issues to understand how synchronization is embedded in modern programming languages.

The primary intended audience is "systems programmers"—the authors of operating systems, library packages, language run-time systems, and server and utility programs. Much of the discussion should also be of interest to application programmers who want to make good use of the synchronization mechanisms available to them, and to computer architects who want to understand the ramifications of their design decisions on systems-level code.

KEYWORDS

Atomicity, barriers, busy-waiting, conditions, locality, locking, memory models, monitors, multiprocessor architecture, nonblocking algorithms, scheduling, semaphores, synchronization, transactional memory.

To Kelly, my wife and partner
of more than 30 years.

Contents

CHAPTER 1

Introduction

In computer science, as in real life, concurrency makes it much more difficult to reason about events. In a linear sequence, if E_1 occurs before E_2, which occurs before E_3, and so on, we can reason about each event individually: E_i begins with the state of the world (or the program) after E_{i-1}, and produces some new state of the world for E_{i+1}. But if the sequence of events $\{E_i\}$ is concurrent with some other sequence $\{F_i\}$, all bets are off. The state of the world prior to E_i can now depend not only on E_{i-1} and its predecessors, but also on some prefix of $\{F_i\}$.

Consider a simple example in which two threads attempt—concurrently—to increment a shared global counter:

```
thread 1:          thread 2:
    ctr++              ctr++
```

On any modern computer, the increment operation (ctr++) will comprise at least three separate machine instructions: one to load ctr into a register, a second to increment the register, and a third to store the register back to memory. This gives us a pair of concurrent instruction sequences:

```
thread 1:          thread 2:
1:  r := ctr       1:  r := ctr
2:  inc r          2:  inc r
3:  ctr := r       3:  ctr := r
```

Intuitively, if our counter is initially 0, we should like it to be 2 when both threads have completed. If each thread executes line 1 before the other executes line 3, however, then both will store a 1, and one of the increments will be "lost."

The problem here is that concurrent sequences of events can *interleave* in arbitrary ways, many of which may lead to incorrect results. In this specific example, only two of the $\binom{6}{3} = 20$ possible interleavings—the ones in which one thread completes before the other starts—will produce the result we want.

Synchronization is the art of precluding interleavings that we consider incorrect. In a distributed (i.e., message-passing) system, synchronization is subsumed in communication: if thread T_2 receives a message from T_1, then in all possible execution interleavings, all the events performed by T_1 prior to its send will occur before any of the events performed by T_2 after its receive. In a shared-memory system, however, things are not so simple. Instead of exchanging messages, threads with shared memory communicate *implicitly* through loads

and stores. Implicit communication gives the programmer substantially more flexibility in algorithm design, but it requires separate mechanisms for explicit synchronization. Those mechanisms are the subject of this monograph.

Significantly, the need for synchronization arises whenever operations are concurrent, regardless of whether they actually run in parallel. This observation dates from the earliest work in the field, led by Edsger Dijkstra [1965; 1968a; 1968b] and performed in the early 1960s. If a single processor core context switches among concurrent operations at arbitrary times, then while some interleavings of the underlying events may be less probable than they are with truly parallel execution, they are nonetheless *possible*, and a correct program must be synchronized to protect against any that would be incorrect. From the programmer's perspective, a multiprogrammed uniprocessor with preemptive scheduling is no easier to program than a multicore or multiprocessor machine.

A few languages and systems guarantee that only one thread will run at a time, and that context switches will occur only at well defined points in the code. The resulting execution model is sometimes referred to as "cooperative" multithreading. One might at first expect it to simplify synchronization, but the benefits tend not to be significant in practice. The problem is that potential context-switch points may be hidden inside library routines, or in the methods of black-box abstractions. Absent a programming model that attaches a true or false "may cause a context switch" tag to every method of every system interface, programmers must protect against unexpected interleavings by using synchronization techniques analogous to those of truly concurrent code.

As it turns out, almost all synchronization patterns in real-world programs (i.e., all conceptually appealing constraints on acceptable execution interleaving) can be seen as instances of either *atomicity* or *condition synchronization*. Atomicity ensures that a specified sequence of instructions participates in any possible interleavings as a single, indivisible unit—that nothing else appears to occur in the middle of its execution. (Note that the very concept of interleaving is based on the assumption that underlying machine instructions are themselves atomic.) Condition synchronization ensures that a specified operation does not occur until some necessary precondition is true. Often, this precondition is the completion of some other operation in some other thread.

Distribution

At the level of hardware devices, the distinction between shared memory and message passing disappears: we can think of a memory cell as a simple process that receives load and store messages from more complicated processes, and sends value and ok messages, respectively, in response. While theoreticians often think of things this way (the annual *PODC* [*Symposium on Principles of Distributed Computing*] and *DISC* [*International Symposium on Distributed Computing*] conferences routinely publish shared-memory algorithms), systems programmers tend to regard shared memory and message passing as fundamentally distinct. This monograph covers only the shared-memory case.

1.1 ATOMICITY

The example on p. 1 requires only atomicity: correct execution will be guaranteed (and incorrect interleavings avoided) if the instruction sequence corresponding to an increment operation executes as a single atomic unit:

```
thread 1:                    thread 2:
    atomic                       atomic
        ctr++                        ctr++
```

The simplest (but not the only!) means of implementing atomicity is to force threads to execute their operations one at a time. This strategy is known as *mutual exclusion*. The code of an atomic operation that executes in mutual exclusion is called a *critical section*. Traditionally, mutual exclusion is obtained by performing acquire and release operations on an abstract data object called a *lock*:

```
lock L
thread 1:                    thread 2:
    L.acquire()                  L.acquire()
        ctr++                        ctr++
    L.release()                  L.release()
```

The acquire and release operations are assumed to have been implemented (at some lower level of abstraction) in such a way that (1) each is atomic and (2) acquire waits if the lock is currently held by some other thread.

In our simple increment example, mutual exclusion is arguably the only implementation strategy that will guarantee atomicity. In other cases, however, it may be overkill. Consider an operation that increments a specified element in an *array* of counters:

```
ctr_inc(i):
    L.acquire()
        ctr[i]++
    L.release()
```

Concurrency and Parallelism

Sadly, the adjectives "concurrent" and "parallel" are used in different ways by different authors. For some authors (including the current one), two operations are *concurrent* if both have started and neither has completed; two operations are *parallel* if they may actually execute at the same time. Parallelism is thus an *implementation of* concurrency. For other authors, two operations are concurrent if there is no correct way to assign them an order in advance; they are parallel if their executions are independent of one another, so that any order is acceptable. An interactive program and its event handlers, for example, are concurrent with one another, but not parallel. For yet other authors, two operations that may run at the same time are considered concurrent (also called *task parallel*) if they execute different code; they are parallel if they execute the *same* code using different data (also called *data parallel*).

If thread 1 calls ctr_inc(i) and thread 2 calls ctr_inc(j), we will need mutual exclusion only if i = j. We can increase potential concurrency with a finer *granularity* of locking—for example, by declaring a separate lock for each counter, and acquiring only the one we need. In this example, the only downside is the space consumed by the extra locks. In other cases, however, fine-grain locking can introduce performance or correctness problems. Consider an operation designed to move n dollars from account i to account j in a banking program. If we want to use fine-grain locking (so unrelated transfers won't exclude one another in time), we need to acquire two locks:

```
move(n, i, j):
    L[i].acquire()
    L[j].acquire()          // (there's a bug here)
        acct[i] −= n
        acct[j] += n
    L[i].release()
    L[j].release()
```

If lock acquisition and release are expensive, we will need to consider whether the benefit of concurrency in independent operations outweighs the cost of the extra lock. More significantly, we will need to address the possibility of *deadlock*:

```
thread 1:                   thread 2:
    move(100, 2, 3)             move(50, 3, 2)
```

If execution proceeds more or less in lockstep, thread 1 may acquire lock 2 and thread 2 may acquire lock 3 before either attempts to acquire the other. Both may then wait forever. The simplest solution in this case is to always acquire the lower-numbered lock first. In more general cases, if may be difficult to devise a static ordering. Alternative atomicity mechanisms—in particular, *transactional memory*, which we will consider in chapter 9—attempt to achieve the concurrency of fine-grain locking without its conceptual complexity.

From the programmer's perspective, fine-grain locking is a means of implementing atomicity for large, complex operations using smaller (possibly overlapping) critical sections. The burden of ensuring that the implementation is correct (that it does, indeed, achieve deadlock-free atomicity for the large operations) is entirely the programmer's responsibility. The appeal of transactional memory is that it raises the level of abstraction, allowing the programmer to delegate this responsibility to some underlying system.

Whether atomicity is achieved through coarse-grain locking, programmer-managed fine-grain locking, or some form of transactional memory, the intent is that atomic regions appear to be indivisible. Put another way, any realizable execution of the program—any possible interleaving of its machine instructions—must be indistinguishable from (have the same externally visible behavior as) some execution in which the instructions of each atomic operation are contiguous in time, with no other instructions interleaved among them. As we shall see in Chapter 3, there are several possible way to formalize this requirement, most notably *linearizability* and several variants on *serializability*.

1.2 CONDITION SYNCHRONIZATION

In some cases, atomicity is not enough for correctness. Consider, for example, a program containing a *work queue*, into which "producer" threads place tasks they wish to have performed, and from which "consumer" threads remove tasks they plan to perform. To preserve the structural integrity of the queue, we will need each insert or remove operation to execute atomically. More than this, however, we will need to ensure that a remove operation executes only when the queue is nonempty and (if the size of the queue is bounded) an insert operation executes only when the queue is nonfull:

```
Q.remove():                                  Q.insert(d):
    atomic                                       atomic
        await !Q.empty()                             await !Q.full()
        // return data from next full slot           // put d in next empty slot
```

In the synchronization literature, a concurrent queue (of whatever sort of objects) is sometimes called a *bounded buffer*; it is the canonical example of mixed atomicity and condition synchronization. As suggested by our use of the await *condition* notation above (notation we have not yet explained how to implement), the conditions in a bounded buffer can be specified at the beginning of the critical section. In other, more complex operations, a thread may need to perform nontrivial work within an atomic operation before it knows what condition(s) it needs to wait for. Since another thread will typically need to access (and modify!) some of the same data in order to make the condition true, a mid-operation wait needs to be able to "break" the atomicity of the surrounding operation in some well-defined way. In Chapter 7 we shall see that some synchronization mechanisms support only the simpler case of waiting at the beginning of a critical section; others allow conditions to appear anywhere inside.

In many programs, condition synchronization is also useful *outside* atomic operations—typically as a means of separating "phases" of computation. In the simplest case, suppose that a task to be performed in thread B cannot safely begin until some other task (data structure initialization, perhaps) has completed in thread A. Here B may spin on a Boolean *flag* variable that is initially false and that is set by A to true. In more complex cases, it is common for a program to go through a *series* of phases, each of which is internally parallel, but must complete in its entirety before the next phase can begin. Many simulations, for example, have this structure. For such programs, a *synchronization barrier*, executed by all threads at the end of every phase, ensures that all have arrived before any is allowed to depart.

It is tempting to suppose that atomicity (or mutual exclusion, at least) would be simpler to implement—or to model formally—than condition synchronization. After all, it could be thought of as a subcase: "wait until no other thread is currently in its critical section." The problem with this thinking is the scope of the condition. By standard convention, we allow conditions to consider only the values of variables, not the states of other threads.

Seen in this light, atomicity is the more demanding concept: it requires agreement among *all* threads that their operations will avoid interfering with each other. And indeed, as we shall see in Section 3.3, atomicity is more difficult to implement, in a formal, theoretical sense.

1.3 SPINNING VS. BLOCKING

Just as synchronization patterns tend to fall into two main camps (atomicity and condition synchronization), so too do their implementations: they all employ *spinning* or *blocking*. Spinning is the simpler case. For condition synchronization, it takes the form of a trivial loop:

```
while !condition
    // do nothing (spin)
```

For mutual exclusion, the simplest implementation employs a special hardware instruction known as test_and_set (TAS). The TAS instruction, available on almost every modern machine, sets a specified Boolean variable to true and returns the previous value. Using TAS, we can implement a trivial *spin lock*:

```
type lock = Boolean := false

L.acquire():                        L.release():
    while !TAS(&L)                      L := false
        // spin
```

Here we have equated the acquisition of L with the act of *changing* it from false to true. The acquire operation repeatedly applies TAS to the lock until it finds that the previous value was *false*. As we shall see in Chapter 4, the trivial test_and_set lock has several major performance problems. It is, however, correct.

The obvious objection to spinning (also known as *busy-waiting*) is that it wastes processor cycles. In a multiprogrammed system it is often preferable to *block*—to yield the processor core to some other, runnable thread. The prior thread may then be run again later—either after some suitable interval of time (at which point it will check its condition, and possibly yield, again), or at some particular time when another thread has determined that the condition is finally true.

The software responsible for choosing which thread to execute when is known as a *scheduler*. In many systems, scheduling occurs at two different levels. Within the operating system, a kernel-level scheduler implements (kernel-level) threads on top of some smaller number of processor cores; within the user-level run-time system, a user-level scheduler implements (user-level) threads on top of some smaller number of kernel threads. At both levels, the code that implements threads (and synchronization) may present a library-style interface, composed entirely of subroutine calls; alternatively, the language in which the

kernel or application is written may provide special syntax for thread management and synchronization, implemented by the compiler.

Certain issues are unique to schedulers at different levels. The kernel-level scheduler, in particular, is responsible for protecting applications from one another, typically by running the threads of each in a different address space; the user-level scheduler, for its part, may need to address such issues as non-conventional stack layout. To a large extent, however, the kernel and runtime schedulers have similar internal structure, and both spinning and blocking may be useful at either level.

While blocking saves cycles that would otherwise be wasted on fruitless re-checks of a condition or lock, it *spends* cycles on the context switching overhead required to change the running thread. If the average time that a thread expects to wait is less than twice the context-switch time, spinning will actually be faster than blocking. It is also the obvious choice if there is only one thread per core, as is sometimes the case in embedded or high-performance systems. Finally, as we shall see in Chapter 7, blocking (otherwise known as *scheduler-based synchronization*) must be built *on top of* spinning, because the data structures used by the scheduler itself require synchronization.

1.4 SAFETY AND LIVENESS

Whether based on spinning or blocking, a correct implementation of synchronization requires both *safety* and *liveness*. Informally, safety means that bad things never happen: we never have two threads in a critical section for the same lock at the same time; we never have all of the threads in the system blocked. Liveness means that good things eventually happen: if lock L is free and at least one thread is waiting for it, some thread eventually acquires it; if queue Q is nonempty and at least one thread is waiting to remove an element, some thread eventually does.

A bit more formally, for a given program and input, running on a given system, safety properties can always be expressed as predicates P on reachable system states S—

Processes, Threads, and Tasks

Like "concurrent," and "parallel," the terms "process," "thread," and "task" are used in different ways by different authors. In the most common usage (adopted here), a *thread* is an active computation that has the potential to share variables with other, concurrent threads. A *process* is a set of threads, together with the address space and other resources (e.g., open files) that they share. A *task* is a well-defined (typically small) unit of work to be accomplished—most often the closure of a subroutine with its parameters and referencing environment. Tasks are passive entities that may be executed by threads. They are invariably implemented at user level. The reader should beware, however, that this terminology is not universal. Many papers (particularly in theory) use "process" where we use "thread." Ada uses "task" where we use "thread." Mach uses "task" where we use "process." And some systems introduce additional words—e.g. "activation," "fiber," "filament," or "hart."

that is, $\forall S[P(S)]$. Liveness properties require at least one extra level of quantification: $\forall S[P(S) \rightarrow \exists T[Q(T)]]$, where T is a subsequent state in the *same execution* as S. From a practical perspective, liveness properties tend to be harder than safety to ensure—or even to define; from a formal perspective, they tend to be harder to prove.

Livelock freedom is one of the simplest liveness properties. It insists that threads not execute forever without making forward progress. In the context of locking, this means that if L is free and thread T has called L.acquire(), there must exist some bound on the number of instructions T can execute before *some* thread acquires L. *Starvation freedom* is stronger. Again in the context of locks, it insists that if every thread that acquires L eventually releases it, and if T has called L.acquire(), there must exist some bound on the number of instructions T can execute before acquiring L itself. Still stronger notions of *fairness* among threads can also be defined, but these are beyond the scope of this monograph.

We will return briefly to the subject of liveness in Section 3.2. Most of our discussions of correctness, however, will focus on safety properties. Interestingly, *deadlock freedom*, which one might initially imagine to be a matter of liveness, is actually one of safety: it simply insists that all reachable states have at least one thread unblocked (for purposes of this definition, a busy-waiting thread is considered to be blocked).

1.5 THE REST OF THIS MONOGRAPH

Chapters 4, 5, and 7 are in some sense the heart of the monograph: they cover spin locks, busy-wait condition synchronization (barriers in particular), and scheduler-based synchronization, respectively. To set the stage for these, Chapter 2 surveys aspects of multicore and multiprocessor architecture that significantly impact the design or performance of synchronization mechanisms, and Chapter 3 introduces formal concepts that illuminate issues of feasibility and correctness.

Between moving from busy-wait to scheduler-based synchronization, Chapter 6 considers atomicity mechanisms that have been optimized for the important special case in which most operations are read-only. Later, Chapter 8 provides a brief introduction to *nonblocking algorithms*, which are carefully designed in such a way that all possible thread interleavings are correct. Chapter 9 provides a similarly brief introduction to *transactional memory*, which uses speculation to implement atomicity without (in typical cases) requiring mutual exclusion. (A full treatment of both of these topics is beyond the scope of the monograph.) Appendix A summarizes the widely used pthread (Posix) synchronization library.

CHAPTER 2

Architectural Background

Both the correctness and the performance of synchronization mechanisms are crucially dependent on certain architectural details of multicore and multiprocessor machines. This chapter provides an overview of these details. It can be skimmed by those already familiar with the subject, but should probably not be skipped in its entirety: the implications of store buffers and directory-based coherence on synchronization algorithms, for example, may not be immediately obvious, and the semantics of memory fence and *read-modify-write* instructions may not be universally familiar.

The chapter is divided into three main sections. In the first, we consider the implications for parallel programs of caching and coherence protocols. In the second, we consider *consistency*—the degree to which accesses to different memory locations can or cannot be assumed to occur in any particular order. In the third, we survey the various read-modify-write instructions—test_and_set and its cousins—that underlie most implementations of atomicity.

2.1 CORES AND CACHES: BASIC SHARED-MEMORY ARCHITECTURE

Figures 2.1 and 2.2 depict two of the many possible configurations of processors, cores, caches, and memories in a modern parallel machine. In a so-called *symmetric* machine, all memory banks are equally distant from every processor core. In a *nonuniform* memory access (NUMA) machine, each memory bank is associated with a processor (or, some cases with a multi-processor *node*), and can be accessed by cores of the local processor more quickly than by cores of other processors.

As feature sizes continue to shrink, the number of cores per processor can be expected to increase. As of this writing, the typical desk-side machine has 1–4 processors with 2–8 cores each. Server-class machines are architecturally similar, but with the potential for many more processors. On some machines, each core may be multithreaded—capable of executing instructions from more than one thread at a time (current per-core thread counts range from 1–8). Each core typically has a private level-1 (L1) cache, and shares a level-2 cache with other cores in its local *cluster*. Clusters on the same processor of a symmetric machine then share a common L3 cache. On a NUMA machine in which the L2 connects directly to the global interconnect, the L3 may sometimes be thought of as "belonging" to the memory.

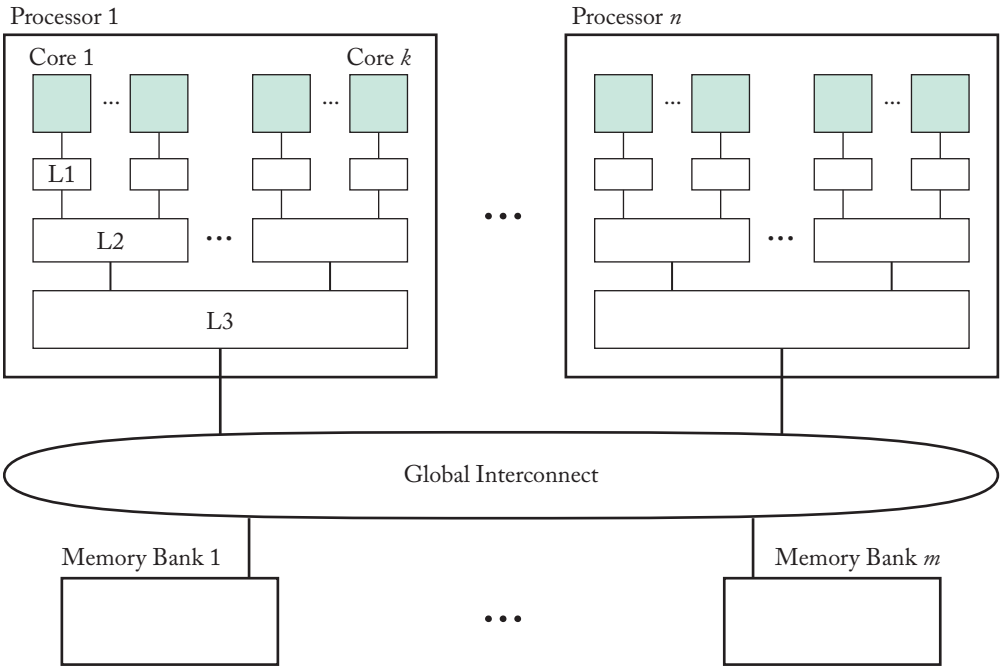

Figure 2.1: Typical symmetric (uniform memory access—UMA) machine. Numbers of components of various kinds, and degree of sharing at various levels, differs across manufacturers and models.

In a machine with more than one processor, the global interconnect may have various topologies. On small machines, broadcast buses and crossbars are common; on large machines, a network of point-to-point links is more common. For synchronization purposes, broadcast has the side effect of imposing a total order on all inter-processor messages; we shall see in Section 2.2 that this simplifies the design of concurrent algorithms—synchronization algorithms in particular. Ordering is sufficiently helpful, in fact, that some large machines (notably those sold by Oracle) employ two different global networks: one for data requests, which are small, and benefit from ordering, and the other for replies, which require significantly more aggregate bandwidth, but do not need to be ordered.

As the number of cores per processor increases, on-chip interconnects—the connections among the L2 and L3 caches in particular—can be expected to take on the complexity of current global interconnects. Other forms of increased complexity are also likely, including, perhaps, additional levels of caching, non-hierarchical topologies, and heterogeneous implementations or even instruction sets among cores.

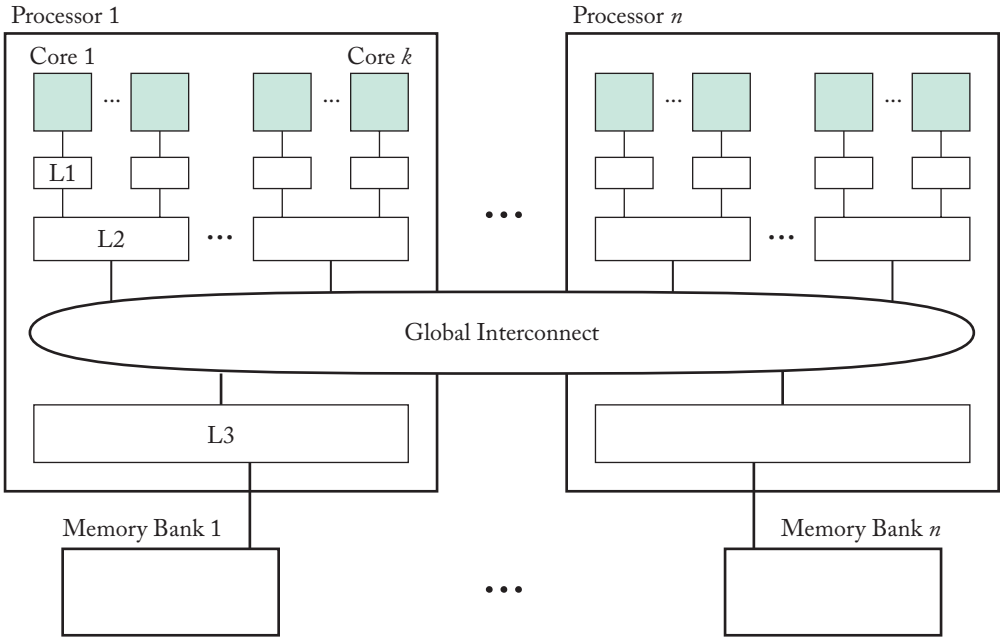

Figure 2.2: Typical nonuniform memory access (NUMA) machine. Again, numbers of components of various kinds, and degree of sharing at various levels, differs across manufacturers and models.

The diversity of current and potential future architectures notwithstanding, multilevel caching has several important consequences for programs on almost any modern machine; we explore these in the following subsections.

2.1.1 TEMPORAL AND SPATIAL LOCALITY

In both sequential and parallel programs, performance can usually be expected to correlate with the temporal and spatial locality of memory references. If a given location l is accessed more than once by the same thread (or perhaps by different threads on the same core or cluster), performance is likely to be better if the two references are close together in time (temporal locality). The benefit stems from the fact that l is likely still to be in cache, and the second reference will be a hit instead of a miss. Similarly, if a thread accesses location l_2 shortly after l_1, performance is likely to be better if the two locations have nearby addresses (spatial locality). Here the benefit stems from the fact that l_1 and l_2 are likely to lie in the same *cache line*, so l_2 will have been loaded into cache as a side effect of loading l_1.

On current machines, cache line sizes typically vary between 32 and 512 bytes. There has been a gradual trend toward larger sizes over time. Different levels of the cache hierarchy may also use different sizes, with lines at lower levels typically being larger.

To improve temporal locality, the programmer must generally restructure algorithms, to change the order of computations. Spatial locality is often easier to improve—for example, by changing the layout of data in memory to co-locate items that are frequently accessed together, or by changing the order of traversal in multidimensional arrays. These sorts of optimizations have long been an active topic of research, even for sequential programs—see, for example, the texts of Muchnick [1997, Chap. 20] or Allen and Kennedy [2002, Chap. 9].

2.1.2 CACHE COHERENCE

On a single-core machine, there is a single cache at each level, and while a given block of memory may be present in more than one level of the memory hierarchy, one can always be sure that the version closest to the top contains up-to-date values. (With a *write-through* cache, values in lower levels of the hierarchy will never be out of date by more than some bounded amount of time. With a *write-back* cache, values in lower levels may be arbitrarily stale—but harmless, because they are hidden by copies at higher levels.)

On a shared-memory parallel system, by contrast—unless we do something special—data in upper levels of the memory hierarchy may no longer be up-to-date if they have been modified by some thread on another core. Suppose, for example, that threads on Cores 1 and k in Figure 2.1 have both been reading variable x, and each has retained a copy in its L1 cache. If the thread on Core 1 then modifies k, even if it writes its value through (or back) to memory, how do we prevent the thread on Core k from continuing to read the stale copy?

A *cache-coherent* parallel system is one in which changes to data, even when cached, are guaranteed to become visible to all threads, on all cores, within a bounded (and typically small) amount of time. On almost all modern machines, coherence is achieved by means of an *invalidation-based cache coherence protocol*. Such a protocol, operating across the system's cache controllers, maintains the invariant that there is at most one writable copy of any given cache line anywhere in the system—and, if the number is one and not zero, there are no read-only copies.

Algorithms to maintain cache coherence are a complex topic, and the subject of ongoing research (for an overview, see the monograph of Sorin et al. [2011], or the more extensive [if dated] coverage of Culler and Singh [1998, Chaps. 5, 6, & 8]). Most protocols in use today descend from the four-state protocol of Goodman [1983]. In this protocol, each line in each cache is either *invalid* (meaning it currently holds no data block), *valid*

(read-only), *reserved* (written exactly once, and up-to-date in memory), or *dirty* (written more than once, and in need of write-back).[1]

To maintain the at-most-one-writable-copy invariant, the coherence protocol arranges, on any write to an *invalid* or *valid* line, to invalidate (evict) any copies of the block in all other caches in the system. In a system with a broadcast-based interconnect, invalidation is straightforward. In a system with point-to-point connections, the coherence protocol typically maintains some sort of *directory* information that allows it to find all other copies of a block.

2.1.3 PROCESSOR (CORE) LOCALITY

On a single-core machine, misses occur on initial accesses and as a result of limited cache capacity or associativity. On a cache-coherent machine, misses may also occur because of *conflicts*. Specifically, a read or write may miss because a previously cached block has been written by some other core, and has reverted to *invalid* state; a write may also miss because a previously *reserved* or *dirty* block has been read by some other core, and has reverted to *valid* state.

Absent program restructuring, conflicts are inevitable if threads on different cores access the same datum (and at least one of them writes it) at roughly the same point in time. In addition to temporal and spatial locality, it is therefore important for parallel programs to exhibit good *thread locality*: as much possible, a given datum should be accessed by only one thread at a time.

Conflicts may also occur when threads are accessing *different* data, if those data happen to lie in the same cache block. This *false sharing* can often be eliminated—yielding a major speed improvement—if data structures are *padded* and *aligned* to occupy an integral number of cache lines. For busy-wait synchronization algorithms, it is particularly impor-

[1] In the cache coherence literature, these states are more commonly known as *invalid*, *shared*, *exclusive*, and *modified*, respectively, as named in the *Illinois protocol* of Papamarcos and Patel [1984]. Written in reverse order, these states yield the acronym MESI, used for the protocol itself.

No-remote-caching Multiprocessors

Most of this monograph assumes a shared-memory multiprocessor with global (distributed) cache coherence, which we have contrasted with machines in which message passing provides the only means of interprocessor communication. There is an intermediate option. Some NUMA machines (notably many of the offerings from Cray, Inc.) support a single global address space, in which any processor can access any memory location, but remote locations cannot be cached. We may refer to such a machine as a no-remote-caching (NRC-NUMA) multiprocessor. Any access to a location in some other processor's memory will traverse the interconnect. Assuming the hardware implements cache coherence *within* each node—in particular, between the local processor(s) and the network interface—memory will still be globally coherent. For the sake of performance, however, system and application programmers will need to employ algorithms that minimize the number of remote references.

---

tant to minimize the extent to which different threads may spin on the same location—or locations in the same cache block. Spinning with a write on a shared location—as we did in the test_and_set lock of Section 1.3, is particularly deadly: each such write leads to interconnect traffic proportional to the number of other spinning threads. We will consider these issues further in Chapter 4.

2.2 CACHE CONSISTENCY

On a single-core machine, it is relatively straightforward to ensure that instructions appear to complete in execution order. Ideally, one might hope that a similar guarantee would apply to parallel machines—that memory accesses, system-wide, would appear to constitute an interleaving (in execution order) of the accesses of the various cores. For many reasons, this sort of *sequential consistency* [Lamport, 1979] is difficult to implement with reasonable performance. Most real machines implement a more *relaxed* (i.e., potentially inconsistent) memory model, in which accesses to different locations by the same thread, or to the same location by different threads, may appear to occur "out of order" from the perspective of threads on other cores. When consistency is required, programmers (or compilers) must insert special *memory fence* instructions (also known as *barriers*) that force the local core to wait for various classes of potentially in-flight events. Such fences are an essential part of most synchronization algorithms.

2.2.1 SOURCES OF INCONSISTENCY

Inconsistency is a natural result of common architectural features. In an *out-of-order* processor, for example—one that can execute instructions in any order consistent with (thread-local) data dependences—a write must be held in the *reorder buffer* until all instructions that precede it in program order have completed. Likewise, since almost any modern processor can generate a burst of store instructions faster than the underlying memory system can absorb them, even writes that are logically ready to commit may need to be buffered for many cycles. The structure that holds these writes is known as a *store buffer*.

Barriers Everywhere

Sadly, the word *barrier* is heavily overloaded. As noted here, it serves as a synonym for *fence*. As noted in Section 1.2 (and explored in more detail in Section 5.2), it is also the name of a synchronization mechanism used to separate program phases. In the programming language community, it refers to code that must be executed when changing a pointer, in order to maintain bookkeeping information for the garbage collector. Finally, as we shall see in Chapter 9, it refers to code that must be executed when reading or writing a shared variable inside an atomic *transaction*, in order to detect and recover from speculation failures. The intended meaning is usually clear from context, but may be confusing to readers who are familiar with only some of the definitions.

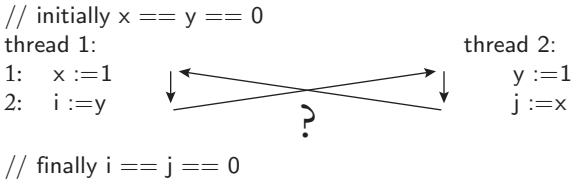

Figure 2.3: An apparent causality loop.

When executing a load instruction, a core checks the contents of its reorder and store buffers before forwarding a request to the memory system. This check ensures that the core always sees its own recent writes, even if they have not yet made their way to cache or memory. At the same time, a load that accesses a location that has *not* been written recently may make its way to memory before logically previous instructions that wrote to other locations. This fact is harmless on a uniprocessor, but consider the implications on a parallel machine, as shown in Figure 2.3. If the write to x is delayed in thread 1's store buffer, and the write to y is similarly delayed in thread 2's store buffer, then both threads may read a zero at line 2, suggesting that line 2 of thread 1 executes before line 1 of thread 2, and line 2 of thread 2 executes before line 1 of thread 1. When combined with program order (line 1 in each thread should execute before line 2 in the same thread), this gives us an apparent "causality loop," which "should" be logically impossible.

Similar problems can occur deeper in the memory hierarchy. A modern machine can require several hundred cycles to service a miss that goes all the way to memory. At each step along the way (core to L1, . . . , L3 to bus, . . .) pending requests may be buffered in a queue. If multiple requests may be active simultaneously (as is common, at least, at the point where we cross the global interconnect), and if some requests may complete more quickly than others, then memory accesses may appear to be reordered. So long as accesses to the *same* location are forced to occur in order, single-threaded code will run correctly. On a multiprocessor, however, sequential consistency may again be violated.

On a NUMA machine, or a machine with a topologically complex interconnect, differing distances among locations provide additional sources of circular "causality." If variable x in Figure 2.3 is close to thread 2 but far from thread 1, and y is close to thread 1 but far from thread 2, the reads on line 2 can easily complete before the writes on line 1, even if all accesses are inserted into the memory system in program order. With a topologically complex interconnect, the cache coherence protocol itself may introduce variable delays—e.g., to dispatch invalidation requests to the various locations that may need to change the state of a local cache line, and to collect acknowledgments. Again, these differing delays may allow line 2 of the example—in both threads—to complete before line 1.

In all the explanations of Figure 2.3, the causality loop results from reads *bypassing* writes—executing in-order (write-then-read) from the perspective of the issuing core, but

out of order (read-then-write) from the perspective of the memory system—or of threads on other cores. On NUMA or topologically complex machines, it may also be possible for reads to bypass reads, writes to bypass reads, or writes to bypass writes. All of these possibilities—if permitted by the hardware—can lead to unintuitive behavior. Examples and additional detail can be found in a variety of sources—notably the work of Adve et al. [1996; 1999].

2.2.2 MEMORY FENCE INSTRUCTIONS

If left unaddressed, memory inconsistency can easily derail attempts at synchronization. Consider the following (quite plausible) programming idiom:

```
// initially x == f == 0
thread 1:                          thread 2:
1:   x := foo()                        while (f == 0)
2:   f := 1                                // spin
3:                                     y := 1/x
```

If foo can never return zero, a programmer might naively expect that thread 2 will never see a divide-by-zero error at line 3. If the write at line 2 in thread 1 can bypass the write in line 1, however, thread 2 may read x too early, and see a value of zero. Similarly, if the read of x at line 3 in thread 2 can bypass the read of f in line 1, a divide-may-zero may again occur, even if the writes in thread 1 complete in order. (While thread 2's read of x is separated from the read of f by a conditional test, the second read may still issue before the first completes, if the branch predictor guesses that the loop will never iterate.)

Memory fence (barrier) instructions allow the programmer to force consistent ordering of memory accesses in situations where it matters, but the hardware might not otherwise guarantee it. The semantically simplest such instruction—a *full fence*—ensures that all effects of all previous instructions are visible to all other threads before allowing any effects of any subsequent instruction to become visible to any other thread. On many machines,

---

Compilers Also Reorder Instructions

While this chapter focuses on architectural issues, it should be noted that compilers also routinely reorder instructions. In any program not written in machine code, compilers perform a variety of optimizations in an attempt to improve performance. Simple examples include reordering computations to expose and eliminate redundancies, hosting invariants out of loops, and "scheduling" instructions to minimize processor pipeline bubbles. Such optimizations are legal so long as they respect control and data dependences within a single thread. Like the hardware optimizations discussed in this section, compiler optimizations can lead to inconsistent behavior when more than one thread is involved. As we shall see in Section 3.4, languages designed for concurrent programming must provide a *memory model* that explains allowable behavior, and some set of primitives—typically the synchronization operations—that function as language-level fences.

a full fence is quite expensive—many tens or even hundreds of cycles. Architects therefore often provide a variety of weaker fences, which prevent some, but not all, inconsistencies.

For purposes of exposition, we treat fences as separate, explicit operations. To indicate the need for a fence that orders previous reads and/or writes with respect to subsequent reads and/or writes, we will use the notation fence(S), where S is some (nonempty) subset of {RR, RW, WR, WW}. In our notation, a fence operation is intended to imply *both* a hardware fence (restricting reordering by the processor implementation) *and* a software fence (restricting reordering by the compiler or interpreter). Compiler writers or assembly langauge programmers interested in porting our pseudocode to some concrete machine will need to restrict their code improvement algorithms accordingly, and also issue appropriate ordering instructions for the hardware at hand. Excellent guidance can be found in Doug Lea's on-line "Cookbook for Compiler Writers" [2001]. Programmers in higher level languages will need to make use of language features (e.g., volatile in Java, or atomic in C++'11) that instruct the compiler to inhibit optimization and issue hardware fences.

To determine the need for fences in the code of a given synchronization operation, we will need to consider both the correctness of the operation itself and the semantics it is intended to provide to the rest of the program. The acquire operation of Peterson's two-thread spin lock [1981], for example, requires internal WW and WR fences in order to achieve mutual exclusion, but these fences are not enough to prevent a thread from reading or writing shared data before it has actually acquired the lock. For that, one needs additional RR and RW fences (code in Sec. 4.1).

Fortunately for most programmers, fences and memory ordering details are generally of concern only to the authors of synchronization mechanisms and low-level concurrent data structures; programmers who use these mechanisms correctly are then typically assured that their programs will behave as if the hardware were sequentially consistent (more on this in Sec. 3.4).

2.2.3 EXAMPLE ARCHITECTURES

A few machines (notably, those employing the c. 1996 MIPS R10000 processor [Yeager, 1996]) have provided sequential consistency, and some researchers have argued that more machines should do so as well [Hill, 1998]. Most machines today, however, fall into two broad classes of more relaxed alternatives. On the SPARC and x86 (both 32- and 64-bit), reads are allowed to bypass writes, but RR, RW, and WW orderings are all guaranteed to be respected by the hardware. Explicit fences are required only when the programmer must ensure that a store completes before a subsequent read. On ARM, POWER, and IA-64 (Itanium) machines, all four combinations of bypassing are possible, and fences must be used when ordering is required.

We will refer to memory models in the SPARC/x86 camp using the SPARC term *TSO* (Total Store Order). We will refer to the other machines as "more relaxed." It should

be emphasized that there are significant differences among machines within a given class—in the available atomic instructions, the behavior of corner cases, and the details of fence instructions. A full explanation is well beyond what we can cover here. For a taste of the complexities involved, see Sewell et al.'s attempt [2010] to formalize the behavior specified informally in Intel's architecture manual [2011, Vol. 3, Sec. 8.2].

On TSO machines, RR, RW, and WW fences can all be elided from our code. On more relaxed machines, they must be implemented with appropriate machine instructions. On some machines, fences are separate instructions; on others their behavior is built into instructions that may serve other purposes as well (in particular, the read-modify-write instructions of Sec. 2.3 often serve as full or partial fences). On some machines, the behavior of weaker fences is defined in terms of the architectural optimizations they inhibit. On other machines, the behavior is defined in terms of the ordering guarantees made to programmers.

One particular pair of fences is perhaps worth special mention. As noted in the previous subsection, a lock acquire operation must ensure that a thread cannot read or write shared data until it has actually acquired the lock. Assuming that acquisition is verified with some sort of load instruction, the appropriate guarantee will be provided by a (RR, RW) fence after the load. In a similar vein, a lock release must ensure that all reads and writes within the critical section have completed before the lock is actually released. Assuming that release is accomplished with some sort of store instruction, the appropriate guarantee will be provided by a (RW, WW) fence before the store. These combinations are common enough that they are sometimes referred to as *acquire* and *release fences*. They are used not only for mutual exclusion, but for most forms of condition synchronization as well. The IA-64 (Itanium) and several research machines—notably the Stanford Dash [Lenoski et al., 1992]—support them directly in hardware. In the mutual exclusion case, they allow work to "migrate" into a critical section both from above (prior to the lock acquire) and from below (after the lock release). They do *not* allow work to migrate *out* of a critical section in either direction.

2.3 ATOMIC PRIMITIVES

To facilitate the construction of synchronization algorithms and concurrent data structures, most modern architectures provide instructions capable of updating (i.e., reading *and* writing) a memory location as a single atomic operation. We saw a simple example—the test_and_set instruction (TAS)—in Section 1.3. A longer list of common instructions appears in Table 2.1.

Originally introduced on mainframes of the 1960s, TAS and Swap are still available on several modern machines, among them the x86 and SPARC. FAA and FAI were introduced for "combining network" machines of the 1980s [Kruskal et al., 1988]. They are uncommon in hardware today, but frequently appear in algorithms in the literature. The semantics

Table 2.1: Common atomic (read-modify-write) instructions.

test_and_set

```
Boolean TAS(Boolean *a):  atomic { t := *a;  *a := true;  return t }
```

swap

```
word Swap(word *a, word w):  atomic { t := *a;  *a := w;  return t }
```

fetch_and_increment

```
int FAI(int *a):  atomic { t := *a;  *a := t + 1;  return t }
```

fetch_and_add

```
int FAA(int *a, int n):  atomic { t := *a;  *a := t + n;  return t }
```

compare_and_swap

```
Boolean CAS(word *a, word old, word new):
    atomic { t := (*a == old);  if (t) *a := new;  return t }
```

load_linked / store_conditional

```
word LL(word *a):  atomic { remember a;  return *a }
Boolean SC(word *a, word w):
    atomic { t := (a is remembered, and has not been evicted since LL)
                if (t) *a := w;  return t }
```

of TAS, Swap, FAI, and FAA should all be self-explanatory. Note that they all return the value of the target location *before* any change was made.

CAS was originally introduced in the c. 1970 IBM 370 architecture [IBM, 1981], and is also found on modern x86, IA-64 (Itanium), and SPARC machines. LL/SC was originally proposed by Jensen et al. for the S-1 AAP Multiprocessor at Lawrence Livermore National Laboratory [Jensen et al., 1987]. It is also found on modern POWER, MIPS, and ARM machines. CAS and LL/SC are *universal* primitives, in a sense we will define formally in Section 3.3. In practical terms, we can use them to build efficient simulations of arbitrary (single-word) read-modify-write (fetch_and_Φ) operations (including all the other operations in Table 2.1).

CAS takes three arguments: a memory location, an old value that is expected to occupy that location, and a new value that should be placed in the location if indeed the old value is currently there. It returns a Boolean value indicating whether the replacement occurred successfully. Given CAS, fetch_and_Φ can be written as follows, for any given function Φ:

```
1:  word fetch_and_Φ(function Φ, word *a):
2:      repeat
3:          old := *a
```

```
4:              new := Φ(old)
5:          until CAS(a, old, new)
6:          return old
```

In effect, this code computes Φ(*a) *speculatively*, and then updates a atomically if its value has not changed since the speculation began. The only way the CAS can fail at line 5 is if some other thread has recently modified a. And in particular, if several threads attempt to perform a fetch_and_Φ on a simultaneously, one of them is guaranteed to succeed, and the system as a whole will make forward progress. This guarantee implies that fetch_and_Φ operations implemented with CAS are *nonblocking* (more specifically, *lock-free*), a property we will consider in more detail in Section 3.2.

One problem with CAS, from an architectural point of view, is that it combines a load and a store into a single instruction, which complicates the implementation of pipelined processors. LL/SC was designed to address this problem. In the fetch_and_Φ idiom above, it replaces the load at line 3 with a special instruction that has the side effect of "tagging" the associated cache line so that the processor will "notice" any subsequent eviction. A subsequent SC will then succeed only if the line is still present in the cache:

```
1:  word fetch_and_Φ(function Φ, word *a):
2:      repeat
3:          old := LL(a)
4:          new := Φ(old)
5:      until SC(a, new)
6:      return old
```

Here any argument for forward progress requires an understanding of why SC might fail. Details vary from machine to machine. In all cases, SC is guaranteed to fail if another thread has modified *a since the LL was performed. On most machines, SC will also fail if a hardware interrupt happens to arrive in the post-LL window. On some machines, it will fail if the cache suffers a capacity or conflict miss, or if the processor mispredicts a branch. To avoid deterministic, spurious failure, the programmer may need to limit (perhaps severely) the types of instructions executed between the LL and SC. If unsafe instructions are required in order to compute the function Φ, one may need a hybrid approach:

```
1:  word fetch_and_Φ(function Φ, word *a):
2:      repeat
3:          old :=*a
4:          new := Φ(old)
5:      until LL(a) == old && SC(a, new)
6:      return old
```

In effect, this code uses LL and SC at line 5 to emulate CAS.

2.3.1 THE ABA PROBLEM

While both CAS and LL/SC appear in algorithms in the literature, the former is quite a bit more common—perhaps because its semantics are self-contained, and do not depend on the implementation-oriented side effect of cache-line tagging. That said, CAS has one significant disadvantage from the programmer's point of view—a disadvantage that LL/SC avoids.

Because it chooses whether to perform its update based on the value in the target location, CAS may succeed in situations where the value has changed (say from A to B) and then changed back again (from B to A) [Treiber, 1986]. In some algorithms, such a change and restoration is harmless: it is still acceptable for the CAS to succeed. In other algorithms, incorrect behavior may result. This possibility, often referred to as the *ABA problem*, is particularly worrisome in pointer-based algorithms. Consider the following (buggy!) code to manipulate a linked-list stack:

```
1: void push(node** top, node* new):
2:   repeat
3:       old := *top
4:       new→next := old
5:   until CAS(top, old, new)
6:
7:
```

```
node* pop(node** top):
    repeat
        old := *top
        if old == null return null
        new := old→next
    until CAS(top, old, new)
    return old
```

Figure 2.4 shows one of many problem scenarios. In (a), our stack contains the elements A and C. Suppose that thread 1 begins to execute pop(&tos), and has completed line 3, but has yet to reach line 6. If thread 2 now executes a (complete) pop(&tos) operation, followed by push(&tos, &B) and then push(&tos, &A), it will leave the stack as shown in (b). If thread 1 now continues, its CAS will succeed, leaving the stack in the broken state shown in (c).

The problem here is that tos changed between thread 1's load and the subsequent CAS. If these two instructions were replaced with LL and SC, the latter would fail—as indeed it should—causing thread 1 to try again.

Emulating CAS

Note that while LL/SC can be used to emulate CAS, the emulation requires a loop to deal with spurious SC failures. This issue was recognized explicitly by the designers of the C++'11 `atomic` types and operations, who introduced two variants of CAS. The `atomic_compare_exchange_strong` operation has the semantics of hardware CAS: it fails only if the expected value was not found. On an LL/SC machine, it is implemented with a loop. The `atomic_compare_exchange_weak` operation admits the possibility of spurious failure: it has the interface of CAS, but is implemented *without* a loop on an LL/SC machine.

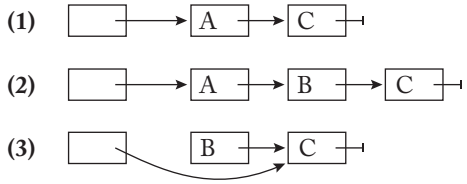

Figure 2.4: The ABA problem in a linked-list stack.

```
1: void push(node** top, node* n):
2:     repeat
3:         ⟨o, c⟩ := *top
4:         n→next := o
5:     until CAS(top, ⟨o, c⟩, ⟨n, c+1⟩)
6:
7:
```

```
node* pop(node** top):
    repeat
        ⟨o, c⟩ := *top
        if o == null return null
        n := o→next
    until CAS(top, ⟨o, c⟩, ⟨n, c+1⟩)
    return o
```

Figure 2.5: Treiber's lock-free stack algorithm, with sequence numbers to solve the ABA problem.

On machines with CAS, programmers must consider whether the ABA problem can arise in the algorithm at hand and, if so, take measures to avoid it. The simplest and most common technique is to devote part of each to-be-CASed word to a sequence number [Treiber, 1986] that is updated on each successful CAS. Using this technique, we can convert our stack code to the (now safe) version shown in Figure 2.5.

The sequence number solution to the ABA problem requires that there be enough bits available for the number that wrap-around cannot occur in any reasonable program execution. Some machines (e.g., the x86 or SPARC when running in 32-bit mode) provide a double-width CAS that is ideal for this purpose. If the maximum word width is required for "real" data, however, another approach may be required.

In many programs, the programmer can reason that a given pointer will reappear in a given data structure only as a result of memory deallocation and reallocation. In a garbage-collected language, deallocation will not occur so long as any thread retains a reference, so all is well. In a language with manual storage management, *hazard pointers* [Herlihy et al., 2002; Michael, 2004a] or *read-copy-update* [McKenney et al., 2001] (Sec. 6.3) can be used to delay deallocation until all concurrent uses of a datum have completed. In the general case (where a pointer can recur without its memory having been recycled), safe CASing may require an extra level of pointer indirection [Jayanti and Petrovic, 2003; Michael, 2004b].

2.3.2 OTHER SYNCHRONIZATION HARDWARE

Several historic machines have provided special locking instructions. The QOLB (queue on lock bit) instruction, originally designed for the Wisconsin Multicube [Goodman et al., 1989], and later adopted for the IEEE Scalable Coherent Interface (SCI) standard [Aboulenein et al., 1994], leverages a coherence protocol that maintains a linked list of copies of a given cache line. When multiple processors attempt to lock the same line at the same time, the hardware arranges to grant the requests in linked-list order. The Kendall Square KSR-1 machine [KSR] provided a similar mechanism based not on an explicit linked list, but on the implicit ordering of nodes in a ring-based network topology. As we shall see in Chapter 4, similar strategies can be emulated in software. The principal argument for the hardware approach is the ability to avoid a costly cache miss when passing the lock (and perhaps its associated data) from one processor to the next [Woest and Goodman, 1991].

The x86 allows any memory-update instruction (e.g., add or increment) to be prefixed with a special LOCK code, rendering it atomic. The benefit to the programmer is limited, however, by the fact that most instructions do not return the previous value from the modified location: two threads executing concurrent LOCKed increments, for example, could be assured that both operations would occur, but could not tell which happened first.

Several supercomputer-class machines have provided special network hardware (generally accessed via memory-mapped I/O) for near-constant-time barrier and "Eureka" operations. These amount to cross-machine AND (all processors are ready) and OR (some processor is ready) computations. We will mention them again in Section 5.2.

In 1991, Stone et al. proposed a multi-word variant of LL/SC, which they called "Oklahoma Update" [Stone et al., 1993] (in reference to the song "All Er Nuthin'" from the Rodgers and Hammerstein musical). Concurrently and independently, Herlihy and Moss proposed a similar mechanism for *Transactional Memory* (TM) [Herlihy and Moss, 1993]. Neither proposal was implemented at the time, but TM enjoyed a rebirth of interest about a decade later. Like queued locks, it can be implemented in software using CAS or LL/SC, but hardware implementations enjoy a substantial performance advantage [Harris et al., 2010]. As of this writing, hardware TM has been implemented on the Azul Systems Vega 2 [Click Jr., 2009], the experimental Sun/Oracle Rock processor [Dice et al., 2009], the IBM Blue Gene/Q, and Intel's "Haswell" version of the x86 (additional implementations are likely to be forthcoming). We will discuss TM in Chapter 9.

CHAPTER 3

Some Useful Theory

Concurrent algorithms and synchronization techniques have a long and very rich history of formalization—far too much to even survey adequately here. Arguably the most accessible resource for practitioners is the text of Herlihy and Shavit [2008]. Deeper, more mathematical coverage can be found in the text of Schneider [1997]. On the broader topic of distributed computing (which as noted in the box on page 2 is viewed by theoreticians as a superset of shared memory concurrency), interested readers may wish to consult the classic textbook by Lynch [1996].

For the purposes of the current text, we provide a brief introduction here to *safety*, *liveness*, the *consensus hierarchy*, and formal *memory models*. Safety and liveness were mentioned briefly in Section 1.4. The former says that bad things never happen; the latter says that good things eventually do. The consensus hierarchy explains the relative expressive power of atomic hardware primitives like test_and_set and compare_and_swap. Memory models explain which writes may be seen by which reads under which circumstances; they help to regularize the "out of order" memory references mentioned in Section 2.2.

3.1 SAFETY

Most concurrent data structures (objects) are adaptations of sequential data structures. Each of these, in turn, has its own *sequential semantics*, typically specified as a set of preconditions and postconditions for each of the methods that operate on the structure, together with *invariants* that all the methods must preserve. The sequential implementation of an object is considered safe if each method, called when its preconditions are true, terminates after a finite number of steps, having ensured the postconditions and preserved the invariants.

When parallelizing a previously sequential object, we typically wish to allow concurrent method calls ("operations"), each of which should appear to occur atomically. This goal in turn leads to at least three safety issues:

1. Because control flow within a thread no longer dictates the order in which operations are called, we cannot in general assume that nontrivial preconditions will always hold. We typically address this problem either by insisting that the operations be *total* (i.e., that the precondition be simply true, so the operation can be performed under any circumstances), or by using condition synchronization to wait until the preconditions hold. The former option is trivial if we are willing to return an indication that the

operation is not currently valid (think, for example, of a dequeue operation that returns a "queue is currently empty" status code). The latter option is explored in Chapter 5.

2. Because threads may wait for one another due to locking or condition synchronization, we must address the possibility of *deadlock*, in which no thread is able to make progress. We consider lock-based deadlock in the first subsection below. Deadlocks due to condition synchronization are a matter of application-level semantics, and must be addressed on a program-by-program basis.

3. The notion of atomicity requires clarification. If operations do not actually execute one at a time in mutual exclusion, we must somehow specify the order(s) in which they are permitted to *appear* to execute. We consider several popular notions of ordering, and the differences among them, in the second subsection below.

3.1.1 DEADLOCK FREEDOM

As noted in Section 1.4, deadlock freedom is a safety property: it requires that there be no reachable state of the system in which all threads are either spinning or blocked in the scheduler. As originally observed by Coffman et al. [1971], deadlock requires four simultaneous conditions:

exclusive use – threads require access to some sort of non-sharable "resources"

hold and wait – threads wait for unavailable resources while continuing to hold resources they have already acquired

irrevocability – resources cannot be forcibly taken from threads that hold them

circularity – there exists a circular chain of threads in which each is holding a resource needed by the next

In shared-memory parallel programs, "non-sharable resources" often correspond to portions of a data structure, with access protected by mutual exclusion ("mutex") locks. Given that exclusive use is fundamental, deadlock can then be addressed by breaking any one of the remaining three conditions. For example:

1. We can break the hold-and-wait condition by requiring a thread that wishes to perform a given operation to request all of its locks at once. This approach is problematic in modular software, or in situations where the identify of some of the locks depends on conditions that cannot be evaluated without holding other locks (suppose, for example, that we wish to move an element atomically from set A to set $f(v)$, where v is the value of the element drawn from set A).

2. We can break the irrevocability condition by requiring a thread to release any locks it already holds when it tries to acquire a lock that is held by another thread. This approach is commonly employed (automatically) in transactional memory systems, which are able to "back a thread out" and retry an operation (transaction) that encounters a locking conflict. It can also be used (more manually) in any system capable of dynamic *deadlock detection* (see, for example, the recent work of Koskinen and Herlihy [2008]). Retrying is complicated by the possibility that an operation may already have generated externally-visible side effects, which must be "rolled back" without compromising global invariants. We will consider rollback further in Chapter 9.

3. We can break the circularity condition by imposing a static order on locks, and requiring that every operation acquire its locks according to that static order. This approach is slightly less onerous than requiring a thread to request all its locks at once, but still far from general. It does not, for example, provide an acceptable solution to the "move from A to $f(v)$" example in strategy 1 above.

Strategy 3 is widely used in practice. It appears, for example, in every major operating system kernel. The lack of generality, however, and the burden of defining—and respecting—a static order on locks, makes strategy 2 quite appealing, particularly when it can be automated, as it is in transactional memory. An intermediate alternative, sometimes used for applications whose synchronization behavior is well understood, is to consider, at each individual lock request, whether there is a feasible order in which currently active operations might complete (under worst-case assumptions about the future resources they might need in order to do so), even if the current lock is granted. The best known strategy of this sort is the *Banker's algorithm* of Dijkstra [early 1960s, 1982], originally developed for the THE operating system [Dijkstra, 1968a]. Where strategies 1 and 3 may be said to *prevent* deadlock by design, the Banker's algorithm is often described as deadlock *avoidance*, and strategy 2 as deadlock *recovery*.

3.1.2 ATOMICITY

In Section 2.2 we introduced the notion of *sequential consistency*, which requires that low-level memory accesses appear to occur in some global total order—i.e., "one at a time"—with each core's accesses appearing in program order (the order specified by the core's sequential program). When considering the order of program-level operations on a concurrent object, it is tempting to ask whether sequential consistency can help. In one sense, the answer is clearly no: correct sequential code will typically not work correctly when executed (without synchronization) by multiple threads concurrently—even on a system with sequentially consistent memory. Conversely, as we shall see in Section 3.4), one can (with appropriate synchronization) build correct high-level objects on top of a system whose memory is more relaxed.

At the same time, the notion of sequential consistency suggests a way in which we might define atomicity for a concurrent object, allowing us to infer what it means for code to be properly synchronized. After all, the memory system is a complex concurrent object from the perspective of a memory architect, who must implement atomic load and store instructions via messages across a distributed cache-cache interconnect. Just as the designer of a sequentially consistent memory system seeks to achieve the appearance of a total order on memory accesses, consistent with per-core program order, so too can the concurrent object designer seek to achieve the appearance of a total order on high-level operations, consistent with the order of each thread's sequential program. In any execution that appears to exhibit such a total order, each operation can be said to have executed atomically.

Sequential Consistency for High-Level Objects

The implementation of a concurrent object O is said to be *sequentially consistent* if, in every possible execution, the operations on O appear to occur in (have the same arguments and return values that they would have had in) some total order that is consistent with program order in each thread. Unfortunately, there is a problem with sequential consistency that limits its usefulness for high-level concurrent objects: lack of composability.

A multiprocessor memory system is, in effect, a single concurrent object, designed at one time by one architectural team. Its methods are the memory access instructions. A high-level concurrent object, by contrast, may be designed in isolation, and then used with other such objects in a single program. Suppose we have implemented object A, and have proven that in any given program, operations performed on A will appear to occur in some total order consistent with program order in each thread. Suppose we have a similar guarantee for object B. We should like to be able to guarantee that in any given program, operations on A and B will appear to occur in some *single* total order consistent with program order in each thread. That is, we should like the operations on A and B to *compose*. Sadly, they may not.

As a simple if somewhat contrived example, consider a replicated integer object, in which threads read their local copy (without synchronization) and update all copies under protection of a lock:

```
// initially L is free and A[i] == 0 ∀ i ∈ 𝒯

void put(int v):
    L.acquire()
    for i ∈ 𝒯
        A[i] := v
    L.release()

int get():
    return A[self]
```

(Throughout this monograph, we use $\mathcal{T}$ to represent the set of thread ids, which we assume to be dense.)

Because of the lock, put operations are fully ordered. Further, because a get operation performs only a single (atomic) access to memory, it is easily ordered with respect to all

puts, following those that have updated the relevant element of A, and preceding those that have not. It is straightforward to identify a total order on operations that respects these constraints and that is consistent with program order in each thread. In other words, our counter is sequentially consistent.

On the other hand, consider what happens if we have *two* counters—call them X and Y. Because get operations can occur "in the middle of" a put at the implementation level, we can imagine a scenario in which threads $T3$ and $T4$ perform gets on X and Y while both objects are being updated—and see the updates in opposite orders:

	local values of shared objects			
	$T1$	$T2$	$T3$	$T4$
	X Y	X Y	X Y	X Y
initially	0 0	0 0	0 0	0 0
$T1$ begins X.put(1)	1 0	1 0	0 0	0 0
$T2$ begins Y.put(1)	1 1	1 1	0 1	0 0
$T3$: X.get() returns 0	1 1	1 1	0 1	0 0
$T3$: Y.get() returns 1	1 1	1 1	0 1	0 0
$T1$ finishes X.put(1)	1 1	1 1	1 1	1 0
$T4$: X.get() returns 1	1 1	1 1	1 1	1 0
$T4$: Y.get() returns 0	1 1	1 1	1 1	1 0
$T2$ finishes Y.put(1)	1 1	1 1	1 1	1 0

At this point, $T3$ thinks the put to Y happened before the put to X, but $T4$ thinks the put to X happened before the put to Y. To solve this problem, we might require the implementation of a shared object to ensure that updates appear to other threads to happen at some single point in time.

But this is not enough. Consider a software emulation of the hardware *write buffers* described in Section 2.2.1. To perform a put on object X, thread T inserts the desired new value into a local queue and continues execution. Periodically, a helper thread drains the queue and applies the updates to the master copy of X, which resides in some global location. To perform a get, T inspects the local queue (synchronizing with the helper as necessary) and returns any pending update; otherwise it returns the global value of X. From the point of view of every thread other than T, the update occurs when it is applied to the global value of X. From T's perspective, however, it happens early, and, in a system with more than one object, we can easily obtain the "bow tie" causality loop of Figure 2.3. This scenario suggests that we require updates to appear to other threads at the same time they appear to the updater—or at least before the updater continues execution.

Linearizability

To address the problem of composability, Herlihy and Wing introduced the notion of *linearizability* [1990]. For more than 20 years it has served as the ordering criterion of choice for high-level concurrent objects. The implementation of object O is said to be linearizable if, in every possible execution, the operations on O appear to occur in some total order that is consistent not only with program order in each thread but also with any ordering that threads are able to observe by other means.

More specifically, linearizability requires that each operation appear to occur instantaneously at some point in time between its call and return. The "instantaneously" part of this requirement precludes the shared counter scenario above, in which $T3$ and $T4$ have different views of partial updates. The "between its call and return" part of the requirement precludes the software write buffer scenario, in which T may see its own updates early.

For the sake of precision, it should be noted that there is no absolute notion of objective time in a parallel system, any more than there is in Einsteinian physics. (For more on the notion of time in parallel systems, see the classic paper by Lamport [1978].) What really matters is observable orderings. When we say that an event must occur at a single instant in time, what we mean is that it must be impossible for thread A to observe that an event has occurred, for A to subsequently communicate with thread B (e.g., by writing a variable that B reads), and then for B to observe that the event has not yet occurred.

When designing a concurrent object, we typically identify a *linearization point* within each method at which a call to that method can be said to have occurred. In the trivial case in which every method is bracketed by acquisition and release of a common object lock, the linearization point can be anywhere inside the method—e.g., at the lock release. In the more general case, it can become significantly more difficult to identify a linearization point, and to prove that everything before that point can be recognized by other threads as merely preparation, and everything after that point as merely cleanup.

In an algorithm based on fine-grain locks, the linearization point of a method may correspond to the release of some particular one of the locks. In a nonblocking algorithm, in which all possible interleavings must be provably correct, the linearization point must correspond to an individual load, store, or other atomic primitive. In the nonblocking stack of Figure 2.5, for example, push and pop both linearize at their final compare_and_swap instruction. In a complex method, there may be multiple possible linearization points, depending on the flow of control at run time. There are even algorithms in which the outcome of a run-time check allows a method to determine that it linearized at some *previous* instruction, earlier in the execution [Harris et al., 2002].

Given linearizable implementations of objects A and B, one can prove that in every possible program execution, the operations on A and B will appear to occur in some *single*

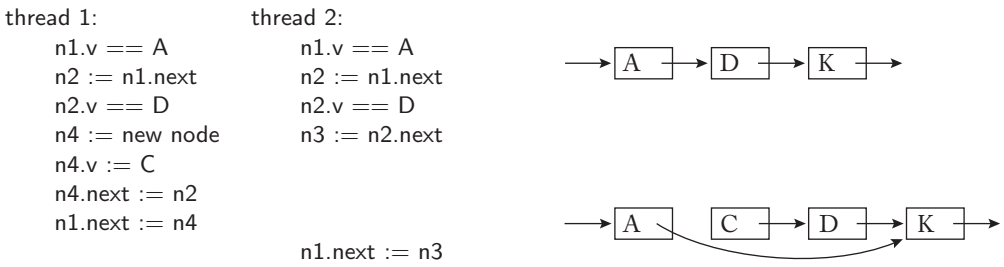

Figure 3.1: Improperly synchronized list updates. This code can lose node C even on a machine with sequential consistency.

total order that is consistent both with program order in each thread and with any other ordering that threads are able to observe. In other words, linearizability is composable.[1]

Hand-over-hand Locking (Lock Coupling). As an example of linearizability achieved through fine-grain locking, consider the task of parallelizing a set abstraction implemented as a sorted, singly-linked list with insert, remove, and lookup operations. Absent synchronization, it is easy to see how the list could become corrupted. In Figure 3.1, the code at left shows a possible sequence of instructions executed by thread 1 in the process of inserting a new node containing the value C, and a concurrent sequence of instructions executed by thread 2 in the process of deleting the node containing the value D. If interleaved as shown, these instructions will transform the list at the upper right into the non-list at the lower right, in which node containing C has been lost.

Clearly a global lock—forcing either thread 1 or thread 2 to complete before the other starts—would linearize the updates and avoid loss of C. It can be shown, however, that linearizability can also be maintained with a fine-grain locking protocol in which each thread holds at most two locks at a time, on adjacent nodes in the list [Bayer and Schkolnick, 1977]. By retaining the right-hand lock while releasing the left-hand and then acquiring the right-hand's successor, a thread ensures that it is never overtaken by another thread during its traversal of the list. In Figure 3.1, thread 1 would hold locks on the nodes containing A and D until done inserting the node containing C. Thread 2 would need these same two locks before reading the content of the node containing A. While threads 1 and 2 cannot make their updates simultaneously, one can "chase the other down the list" to the point where the updates are needed, achieving substantially higher concurrency than is possible with a global lock. Similar "hand-over-hand" locking techniques are widely used in concurrent trees and other pointer-based data structures.

[1] In early papers, composability was known as *locality* [Herlihy and Wing, 1990; Weihl, 1989]. The linearizability of a given object O was said to be a local property because it depended only on the implementation of O, not on the other objects with which O might be used in some larger program.

Serializability

Recall that the purpose of an ordering criterion is to clarify the meaning of atomicity. By requiring an operation to complete—and to be visible to all other threads—before it returns to its caller, linearizability guarantees that the order of operations on any given concurrent object will be consistent with all other observable orderings in an execution, including those of other concurrent objects.

The flip side of this guarantee is that linearizability cannot be used to reason about composite operations—ones that manipulate more than one object, but are intended to execute as a single atomic unit.

Consider a banking system in which thread 1 transfers $100 from account A to account B, while thread 2 adds the amounts in the two accounts:

```
// initially A.balance() == B.balance() == 500
thread 1:                          thread 2:
    A.withdraw(100)
                                       sum := A.balance()      // 400
                                       sum +:= B.balance()     // 900
    B.deposit(100)
```

If we think of A and B as separate objects, then the execution can linearize as suggested above, but thread 2 will see a cross-account total that is $100 "too low." If we wish to treat the code in each thread as a single atomic unit, we must disallow this execution—something that neither A nor B can do on its own.

Multi-object atomic operations are the hallmark of database systems, which refer to them as *transactions*. Transactional memory (the subject of Chapter 9) adapts transactions to shared-memory parallel computing, allowing the programmer to request that a multi-object operation like thread 1's transfer or thread 2's sum should execute atomically.

The simplest ordering criterion for database transactions is known as *serializability*. Transactions are said to serialize if they have the same effect they would have had if executed one at a time in some total order. For transactional memory (and sometimes for databases as well), we can extend the model to allow a thread to perform a series of transactions, and require that the global order be consistent with program order in each thread.

It turns out to be NP-hard to determine whether a given set of transactions (with the given inputs and outputs) is serializable [Papadimitriou, 1979]. Fortunately, we seldom need to make such a determination in practice. Generally all we really want is to ensure that the current execution will be serializable—something we can achieve with conservative (sufficient but not necessary) measures. A global lock is a trivial solution, but admits no concurrency. Databases and most TM systems employ more elaborate fine-grain locking. A few TM systems employ nonblocking techniques.

If we regard the objects to be accessed by a transaction as "resources" and revisit the conditions for deadlock outlined at the beginning of Section 3.1.1, we quickly realize that a transaction may, in the general case, need to access some resources before it knows

which others it will need. Any implementation of serializability based on fine-grain locks will thus entail not only "exclusive use," but also both "hold and wait" and "circularity." To address the possibility of deadlock, a database or lock-based TM system must be prepared to break the "irrevocability" condition by releasing locks, rolling back, and retrying conflicting transactions.

Like branch prediction or compare_and_swap-based fetch_and_Φ, this strategy of proceeding "in the hope" that things will work out (and recovering when they don't) is an example of *speculation*. So-called *lazy* TM systems take this even further, allowing conflicting (non-serializable) transactions to proceed in parallel until one of them is ready to *commit*—and only then *aborting* and rolling back the others.

Two-Phase Locking. As an example of fine-grain locking for serializability, consider a simple scenario in which transactions 1 and 2 read and update symmetric variables:

```
// initially x == y == 0
transaction 1:          transaction 2:
    t1 := x                 t2 := y
    y++                     x++
```

Left unsynchronized, this code could result in t1 == t2 == 0, even on a sequentially consistent machine—something that should not be possible if the transactions are to serialize. A global lock would solve the problem, but would be far too conservative for transactions larger than the trivial ones shown here. If we associate fine-grain locks with individual variables, we still run into trouble if thread 1 releases its lock on x before acquiring the lock on y, and thread 2 releases its lock on y before acquiring the lock on x.

It turns out [Eswaran et al., 1976] that serializability can always be guaranteed if threads acquire all their locks (in an "expansion phase") before releasing any of them (in a "contraction phase"). As noted above, this convention admits the possibility of deadlock. In our example, transaction 1 might lock x and transaction 2 lock y before either attempts to acquire the other. To detect the problem and trigger rollback, a system based on two-phase locking may construct and maintain a dependence graph at run time. Alternatively (and more conservatively), it may simply limit the time it is willing to wait for locks, and assume the worst when this timeout is exceeded.

Strict Serializability

The astute reader may have noticed the strong similarity between the definitions of sequential consistency (for high-level objects) and serializability (with the extension that allows a single thread to perform a series of transactions). The difference is simply that transactions need to be able to access a dynamically chosen set of objects, while sequential consistency is limited to a predefined set of single-object operations.

The similarity between sequential consistency and serializability leads to a common weakness: the lack of required consistency with other orders that may be observed by

a thread. It was by requiring such "real-time" ordering that we obtained composability for single-object operations in the definition of linearizability. Real-time ordering is also important for its own sake in many applications. Without it we might, for example, make a large deposit to a friend's bank account, tell the person we had done so, and yet still encounter an "insufficient funds" message in response to a (subsequent!) withdrawal request. To avoid such counterintuitive scenarios, many database systems—and most TM systems—require *strict serializability*, which is simply ordinary serializability augmented with real-time order: transactions are said to be strictly serializable if they have the same effect they would have had if executed one at a time in some total order that is consistent with program order (if any) in each thread, and with any other order the threads may be able to observe. In particular, if transaction A finishes before transaction B begins, then A must appear before B in the total order.

Relationships Among the Ordering Criteria

Figure 3.1 summarizes the relationships among sequential consistency (for high-level objects), linearizability, serializability, and strict serializability. A system that correctly implements any of these four criteria will provide the appearance of a total order on operations, consistent with per-thread program order. Linearizability and strict serializability add consistency with "real-time" order. Serializability and strict serializability add the ability to work with multi-object atomic operations, but at the expense of composability: an execution that is serializable (or sequentially consistent, or strictly serializable) with respect to each of the objects it uses, individually, is not necessarily serializable with respect to all of them together.

To avoid confusion, it should be noted that "composability" has a different meaning in the database and TM communities from the one presented here. We will consider this alternative meaning in Chapter 9. There the question will be: given several atomic (serializable) operations, can we wrap calls to them inside some larger atomic operation? This style of composability is straightforward to provide in a system based on speculation; it is invariably supported by database and TM systems. It cannot be supported, in the general case, by conservative locking strategies.

3.2 LIVENESS

Safety properties—the subject of the previous section—ensure that bad things never happen: threads are never deadlocked; atomicity is never violated; invariants are never broken. To say that code is correct, however, we generally want more: we want to ensure forward progress. Just as we generally want to know that a sequential program will produce a correct answer in bounded time (not just fail to produce an incorrect answer), we generally want to know that invocations of concurrent operations will complete their work and return.

Table 3.1: Properties of standard ordering criteria.
SC = sequential consistency; L = linearizability;
S = serializability; SS = strict serializability.

	SC	L	S	SS
Equivalent to a sequential order	+	+	+	+
Respects program order in each thread	+	+	+	+
Consistent with other ordering ("real time")	−	+	−	+
Can touch multiple objects atomically	−	−	+	+
Composes	−	+	−	−

An object method is said to be *blocking* if there is some reachable state of the system in which a thread that has called the method will be unable to make forward progress (and return from the method) until some other thread takes action. Lock-based algorithms are inherently blocking: a thread that holds a lock precludes progress on the part of any other thread that needs the same lock. Liveness proofs for lock-based algorithms require not only that the code be deadlock-free, but also that critical sections be free of infinite loops.

A method is said to be *nonblocking* if there is no reachable state of the system that prevents it from completing and returning. Nonblocking algorithms have the desirable property that inopportune preemption (e.g., of a lock holder) never precludes forward progress in other threads. In some environments (e.g., a system with high fault tolerance requirements), nonblocking algorithms may also allow the system to survive when a thread crashes or is prematurely killed. We consider several variants of nonblocking progress in the first subsection below.

In both blocking and nonblocking algorithms, we may also care about *fairness*—the relative rates of progress of different threads. We consider this topic briefly in the second subsection below.

3.2.1 NONBLOCKING PROGRESS

Given the difficulty of guaranteeing any particular rate of execution (in the presence of timesharing, cache misses, page faults, and other sources of variability), we generally speak of progress in terms of abstract program *steps* rather than absolute time.

A method is said to be *wait free* (the strongest variant of nonblocking progress) if it is guaranteed to complete in some bounded number of its own program steps. (This bound need not be statically known.) A method M is said to be *lock free* (a somewhat weaker variant) if *some* thread is guaranteed to make progress (complete an operation on the same object) in some bounded number of M's program steps. M is said to be *obstruction free* (the weakest variant of nonblocking progress) if it is guaranteed to complete in some bounded number of program steps if no other thread executes any steps during that same interval.

Wait freedom is sometimes referred to as *starvation freedom*: a given thread is never prevented from making progress. Lock freedom is sometimes referred to as *livelock freedom*: an individual thread may starve, but the system as a whole is never prevented from making forward progress (equivalently: no set of threads can actively prevent each other from making progress indefinitely). Obstruction-free algorithms can suffer not only from starvation but also from livelock; if all threads but one "hold still" long enough, however, the one running thread is guaranteed to make progress.

Many practical algorithms are lock free or obstruction free. Treiber's stack, for example (Section 2.3.1), is lock-free, as is the widely used queue of Michael and Scott (Section 8.1). Obstruction freedom was first described in the context of Herlihy et al.'s double-ended queue [2003a]. It is also provided by several TM systems (among them the DSTM of Herlihy et al. [2003b], the ASTM of Marathe et al. [2005], and the work of Marathe and Moir [2008]). Moir and Shavit [2005] provide an excellent survey of concurrent data structures, including coverage of nonblocking progress. Sundell and Tsigas [2008] describe a library of such data structures.

Wait-free algorithms are significantly less common. Herlihy [1991] demonstrated that any sequential data structure can be transformed, automatically, into a wait-free concurrent version, but the construction is highly inefficient. Recent work by Kogan and Petrank [2012] (building on a series of earlier results) has shown how to reduce the time overhead dramatically, though space remains proportional to the maximum number of threads in the system.

Most wait-free algorithms—and many lock-free algorithms—employ some variant of *helping*, in which a thread that has stalled after reaching its linearization point, but before completing all its cleanup work, may be assisted by other threads, which need to "get it out of the way" so they can perform their own operations. Other lock-free algorithms—and even a few wait-free algorithms—are able to make do without helping. As a simple example, consider the following implementation of a wait-free counter:

```
initially C[i] == 0 ∀ i ∈ 𝒯

void inc():
    C[self]++

int val():
    rtn := 0
    for i in [1..N]
        rtn +:= C[i]
    return rtn
```

Here the aggregate counter value is taken to be the sum of a set of per-thread values. Because each per-thread value is monotonically increasing, so is the aggregate sum. Given this observation, one can prove that the value returned by the val method will have been correct sometime between its call and its return: it will be bounded below by the number of inc operations that can be determined to have returned before val was called, and above by the number that *cannot* be determined *not* to have yet been called when val returns. In other words, val is linearizable, though its linearization point cannot in general be statically

determined. Because both inc and val comprise a bounded (in this case, statically bounded) number of program steps, the methods are wait free.

3.2.2 FAIRNESS

It is tempting to think of wait freedom as the ultimate form of fairness: no thread ever waits for any other. This thought, however, assumes a fundamental underlying property. Known to theoreticians as *unconditional fairness*, it asserts that any thread that is not blocked eventually executes another program step. In practical terms, unconditional fairness depends on appropriate scheduling at multiple levels of the system—in the hardware, operating system, run-time system, and language implementation—all of which must ensure that runnable threads continue to run.

When threads may block for mutual exclusion or condition synchronization, fairness becomes more problematic. In almost all cases we will want to insist on *weak fairness*, which requires that any thread waiting for a condition that is continuously true (or a lock that is continuously available) eventually executes another program step. In the following program fragment, for example, weak fairness precludes an execution in which thread 1 spins forever: thread 2 must eventually notice that f is false, complete its wait, and set f to true, after which thread 1 must notice the change to f and complete.

```
initially f == false

thread 1:                          thread 2:
    await (f)                          await (!f)
                                       f := true
```

Here we have used the notation await (*condition*) as shorthand for

```
while (not condition)
    // spin
fence(RR, RW)
```

Many more stringent definitions of fairness are possible. In particular, *strong fairness* requires that any thread waiting for a condition that is true infinitely often (or a lock that is available infinitely often) eventually executes another program step. In the following program fragment, for example, strong fairness again precludes an execution in which thread 1 spins forever: thread 2 must notice one of the "windows" in which g is true, complete its wait, and set f to true, after which thread 1 must notice the change and complete.

```
initially f == g == false

thread 1:                          thread 2:
    while (!f)                         await (g)
        g := true                      f := true
        g := false
```

Strong fairness is difficult to truly achieve: it may, for example, require a scheduler to re-check every awaited condition whenever one of its constituent variables is changed, to

make sure that any thread at risk of starving is given a chance to run. Any deterministic strategy that considers only a subset of the waiting threads on each state change risks the possibility of deterministically ignoring some unfortunate thread every time it is able to run.

Fortunately, statistical "guarantees" typically suffice in practice. By considering a randomly chosen thread—instead of all threads—when a scheduling decision is required, we can drive the probability of starvation arbitrarily low. A truly random choice is difficult, of course, but various pseudorandom approaches appear to work quite well. At the hardware level, interconnects and coherence protocols are designed to make it highly unlikely that a race between two cores (e.g., when performing near-simultaneous compare_and_swap instructions) will always be resolved the same way. Within the operating system, run-time system, or language implementation, one can "randomize" the interval between checks of a condition using a pseudorandom number generator or even the natural "jitter" in execution time of nontrivial code blocks on complex modern cores.

Weak and strong fairness address worst-case behavior, and allow executions that still seem grossly unfair from an intuitive perspective (e.g., executions in which one thread succeeds a million times more often than another). Statistical "randomization," by contrast, may achieve intuitively very fair behavior without absolutely precluding worst-case starvation.

In subsequent chapters we will consider synchronization algorithms that range from highly unfair (e.g., test_and_set locks that work well only when there are periodic quiescent intervals, when the lock is free and no thread wants it), to strictly first-come, first-served (e.g., locks in which a thread employs a wait-free protocol to join a FIFO queue). We will also consider intermediate options, such as locks that deliberately balance locality (for performance) against fairness.

Much of the theoretical groundwork for fairness was laid by Nissim Francez [1986]. Proofs of fairness are typically based on *temporal logic*, which provides operators for concepts like "always" and "eventually." A brief introduction to these topics can be found in the text of Ben-Ari [2006, **Chapter 4**]; much more extensive coverage can be found in Schneider's comprehensive work on the theory of concurrency [1997].

3.3 THE CONSENSUS HIERARCHY

In Section 2.3 we noted that CAS and LL/SC are *universal* atomic primitives—capable of implementing arbitrary single-word fetch_and_Φ operations. We suggested—implicitly, at least—that they are fundamentally more powerful than simpler primitives like TAS, Swap, FAI, and FAA. Herlihy formalized this notion of relative power in his work on wait-free synchronization [1991], previously mentioned in Section 3.2.1. The formalization is based on the notion of *consensus*.

Originally formalized by Fischer, Lynch, and Paterson [1985] in a distributed setting, the *consensus problem* involves a set of potentially unreliable threads, each of which "proposes" a value. The goal is for the reliable threads to agree on one of the values they proposed—a task the authors proved to be impossible with asynchronous messages. Herlihy adapted the problem to the shared-memory setting, where powerful atomic primitives can circumvent impossibility. Specifically, Herlihy suggested that such primitives (or, more precisely, the objects on which those primitives operate) be classified according the number of threads for which they can achieve *wait-free* consensus.

It is easy to see that a TAS object can achieve wait-free consensus for two threads:

```
// initially L == 0; proposal[0] and proposal[1] are immaterial
agree(i):
    proposal[self] := i
    fence(WR, WW)
    if TAS(L) return i
    else return proposal[1-self]
```

Herlihy was able to show that this is the best one can do: TAS cannot achieve wait-free consensus for more than two threads. Moreover ordinary loads and stores cannot achieve wait-free consensus at all—even for only two threads. An object supporting CAS, on the other hand (or equivalently LL/SC), can achieve wait-free consensus for an arbitrary number of threads:

```
// initially v == ⊥
agree(i):
    if CAS(&v, ⊥, i) return i
    else return v
```

One can, in fact, define an infinite hierarchy of atomic objects, where those appearing at level k can achieve wait-free consensus for k threads but no more. Objects supporting CAS or LL/SC are said to have *consensus number* ∞. Many other common primitives—including TAS, swap, FAI, and FAA—have consensus number 2. One can define atomic objects at intermediate levels of the hierarchy, but these are not typically encountered on real hardware.

3.4 MEMORY MODELS

As described in Section 2.2, most modern multicore systems are *coherent* but not *sequentially consistent*: changes to a given variable are serialized, and eventually propagate to all cores, but accesses to different locations may appear to occur in different orders from the perspective of different threads—even to the point of introducing apparent causality loops. For programmers to reason about such a system, we need a *memory model*—a formal characterization of its behavior.

There is a rich and extensive literature on memory models. Good starting points include the tutorial of Adve and Gharachorloo [1996] and the articles introducing the models of Java [Manson et al., 2005] and C++ [Boehm and Adve, 2008]. Details vary considerably from one model to another, but most now share a similar framework.

3.4.1 FORMAL FRAMEWORK

Informally, a memory model specifies, for any given program execution, which values (i.e., which writes) may be seen by any given read. More precise definitions depend on the notion of an abstract *program execution*.

Programming language semantics are typically defined in terms of execution on some abstract machine, with language-appropriate built-in types, control-flow constructs, etc. For a given source program and input, a program execution is a set of sequences, one per thread, of loads, stores, and other atomic steps, each of which inspects and/or changes the state of the abstract machine (i.e., memory). Language semantics can be seen as a mapping from programs and inputs to (possibly infinite) sets of valid executions of the language's abstract machine. They can also be seen as a predicate: given a program, its input, and an abstract execution, is the execution valid?

A language implementation provides a similar mapping from programs and inputs to sets of *concrete executions* on some real, physical machine. A language implementation is *safe* if for every well formed source program and input, the concrete executions it produces correspond to (produce the same output as) some valid abstract execution of that program on that input (Figure 3.2). The implementation is *live* if produces at least one such concrete execution whenever the set of abstract executions is nonempty. (We focus here on safety.)

In this execution-based framework, a memory model is the portion of language semantics that determines whether the values read by the loads in an abstract execution are

---

Consensus and Mutual Exclusion

A solution to the consensus problem clearly suffices for "one-shot" mutual exclusion (a.k.a. *leader election*): each thread proposes its own id, and the agreed-upon value indicates which thread is able to enter the critical section. Consensus is not *necessary* however: the winning thread needs to know that it can enter the critical section, but other threads only need to know that they have lost—they don't need to know who won. TAS thus suffices to build a wait-free *try lock* (one whose acquire method returns immediately with a success or failure result) for an arbitrary number of threads.

It is tempting to suggest that any solution to the mutual exclusion problem will suffice to build a *blocking* solution to the consensus problem (have the winner of the competition for the lock write its id to a well-known location; have the losers spin until that value appears), but this is not acceptable: consensus requires agreement among correct threads even in the presence of faulty threads. If a thread acquires the lock and then dies before writing down its id, the others will never reach agreement. This is the beauty of CAS or LL/SC: it allows a thread to win the competition *and* write down its value in a single indivisible step.

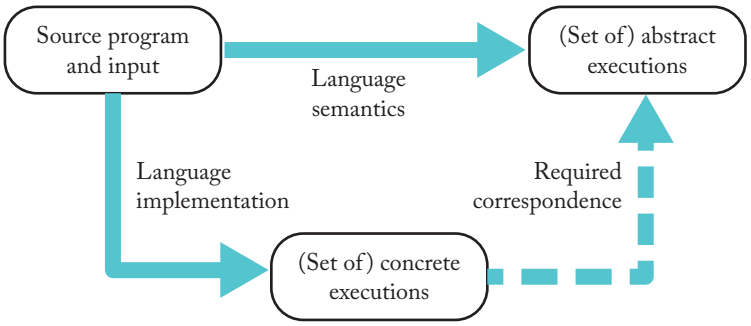

Figure 3.2: Program executions, semantics, and implementations. A valid implementation must produce only those concrete executions whose output agrees with that of some abstract execution allowed by language semantics.

valid, given the values written by the stores. In a single-threaded program, the memory model is trivial: there is a total order on program steps, and each load reads the value written by the most recent store to the same location—or the initial value (if any) if there is no such store.

In a multithreaded program, a load in one thread may read a value written by a store in a different thread. An execution is said to be sequentially consistent if there exists a total order that explains all values read, as in the single-threaded case. But not all executions are sequentially consistent. In the general case, the best we can manage is a partial order known as *happens-before*. To define this order, most memory models begin by distinguishing between "ordinary" and "synchronization" steps in the program execution. Ordinary steps are typically loads and stores. Depending on the language, synchronization steps might be lock acquire and release operations, transactions, monitor entry and exit, message send and receive, or any of a large number of other possibilities. Whatever these steps might be, we use them to proceed as follows:

Program order is a union of disjoint total orders, one per thread.

Synchronization order is a total order, across all threads, on all synchronization steps in the abstract execution. This order must be consistent with program order within each thread. It may also need to satisfy certain language-specific constraints—e.g., acquire and release operations on any given lock may need to occur in alternating order. Crucially, synchronization order is not specified by the source program. An execution is valid only if there exists a synchronization order that leads to a reads-see-writes function that explains the values read.

Synchronizes-with order is a subset of synchronization order induced by language semantics. In a language based on transactional memory, the subset may be trivial: all

transactions are globally ordered. In a language based on locks, each release operation may synchronize with the next acquire of the same lock in synchronization order, but other synchronization steps may be unordered.

Happens-before order is the transitive closure of program order and synchronizes-with order.

3.4.2 DATA RACES

Two ordinary steps (memory accesses) are said to *conflict* if (1) they occur in separate threads, (2) they access the same location, and (3) at least one of them is a store. A program is said to have a *data race* if, for some input, it has a sequentially consistent execution in which two conflicting ordinary steps are adjacent in the total order. Data races are problematic because we don't normally expect an implementation to force the order found in the sequentially consistent execution. This suggests the existence of a concrete execution corresponding to a different abstract execution—one in which all prior steps are the same but the conflicting steps are reversed. It is easy to construct examples (e.g., as suggested in Figure 2.3) in which the remainder of this second execution cannot be sequentially consistent.

Given the definitions of the previous subsection, we can also say that an abstract execution has a data race if, for some valid synchronization order, it contains a pair of conflicting steps that are not ordered by happens-before. A program then has a data race if, for some input, it has an execution containing a data race. This definition turns out to be equivalent to the one based on sequentially consistent executions.

Synchronization Races

The definition of a data race is designed to capture cases in which program behavior may depend on the order in which two ordinary accesses occur, and this order is not constrained by synchronization. In a similar fashion, we may wish to consider cases in which program behavior depends on the outcome of synchronization operations.

For each form of synchronization operation, we can define a notion of conflict. Acquire operations on the same lock, for example, conflict with one another, while an acquire and a release to not, nor do operations on different locks. A program is said to have a *synchroniztion race* if it has a sequentially consistent execution in which two conflicting synchronization operations are adjacent in the total order. Together, data races and synchronization races constitute the class of *general races*.

Because we assume the existence of a total order on synchronization operations, synchronization races never compromise sequential consistency. Rather they provide the means of controlling and exploiting nondeterminism in parallel programs. In any case where we wish to allow conflicting high-level operations to occur in arbitrary order, we design a synchronization race into the program to mediate the conflict.

In a program without any data races, the reads-see-writes function is straightforward: the lack of unordered conflicting accesses implies that no load has more than one most recent prior store in happens-before order. It must therefore read the value written by that store—or the initial value if there is no such store. More formally, one can prove that all executions of a data-race-free program are sequentially consistent. Moreover, since the (first) definition of a data race was based only on sequentially consistent executions, we can provide the programmer with a set of rules that, if followed, will always lead to sequentially consistent executions, with no need to reason about possible relaxed behavior of the underlying hardware. Such a set of rules is said to constitute a *programmer-centric* memory model [Adve and Hill, 1990]. In effect, a programmer-centric model is a contract between the programmer and the implementation: if the programmer follows the rules (i.e., write data-race-free programs), the implementation will provide the illusion of sequential consistency.

But what about programs that *do* have data races? Some researchers have argued that such programs are simply buggy, and should have undefined behavior. This is the approach adopted by C++ [Boehm and Adve, 2008]. It rules out certain categories of programs (e.g., chaotic relaxation [Chazan and Miranker, 1969]), but the language designers had little in the way of alternatives: in the absence of type safety it is nearly impossible to limit the potential impact of a data race. The resulting model is quite simple: if a C++ program has a data race on a given input, its behavior is undefined; otherwise, it follows one of its sequentially consistent executions.

Unfortunately, in a language like Java, even buggy programs must have well defined behavior, to safeguard the integrity of the virtual machine (which may be embedded in some larger, untrusting system). The obvious approach is to say that a load may read the value written by the most recent store on any backward path through the happens-before graph, or by any *incomparable* store (one that is unordered with respect to the load). Unfortunately, as described by Manson et al. [2005], this approach is overly restrictive: it precludes the use of several important compiler optimizations. The actual Java model defines a notion of "incremental justification" that may allow a load to read a value that might have been written by an incomparable store in some *other* hypothetical execution. The details are surprisingly subtle and complex.

3.4.3 REAL-WORLD MODELS

As of this writing, Java and C++ are the only widely used programming languages whose memory models have been precisely specified. C# is assumed to provide a model similar to that of Java. The next official version of C is expected to "reverse-inherit" a variant of the model from C++.

Ada [Ichbiah et al., 1991] was the first language to introduce an explicitly relaxed (if informally specified) memory model. It was designed to facilitate implementation on both

shared-memory and distributed hardware: variables shared between threads were required to be consistent only in the wake of explicit message passing (rendezvous). The reference implementations of several scripting languages (notably Ruby and Python) are sequentially consistent, though other implementations [JRuby; Jython] are not.

A group including representatives of Intel, Oracle, IBM, and Red Hat has proposed transactional extensions to C++ [Adl-Tabatabai et al., 2012]. In this proposal, begin_transaction and end_transaction markers contribute to the happens-before order inherited from standard C++. So-called *relaxed* transactions are permitted to contain other synchronization operations (e.g., lock acquire and release); *atomic* transactions are not. Dalessandro et al. [2010b] have proposed an alternative model in which atomic blocks are fundamental, and other synchronization mechanisms (e.g., locks) are built on top of them.

If we wish to allow programmers to create new synchronization mechanisms or nonblocking data structures (and indeed if any of the built-in synchronization mechanisms are to be written in high-level code, rather than assembler), then the memory model must define synchronization steps that are more primitive than lock acquire and release. Java allows a variable to be labeled volatile, in which case loads and stores that access it are included in the global synchronization order, with each load inducing a synchronizes-with arc (and thus a happens-before arc) from the (unique) preceding store to the same location. C++ provides a substantially more complex facility, in which variables are labeled atomic, and an individual load, store, or fetch_and_Φ operation can be labeled as an acquire fence, a release fence, both, or neither (it can also be labeled as sequentially consistent).

A crucial goal in the design of any practical memory model is to preserve, as much as possible, the freedom of compiler writers to employ code improvement techniques originally developed for sequential programs. The ordering constraints imposed by synchronization operations necessitate not only hardware-level memory fences, but also software-level "compiler fences," which inhibit the sorts of code motion traditionally used for latency tolerance, redundancy elimination, etc. (Recall that in our pseudocode, fence operations are intended to enforce both hardware and compiler ordering.) Much of the complexity of C++ atomic variables stems from the desire to avoid unnecessary hardware and compiler fences. Within reason, programmers should attempt in C++ to specify the minimal ordering constraints required for correct behavior. At the same time, they should resist the temptation to "get by" with minimal ordering in the absence of a solid correctness argument. Recent work by Attiya et al. [2011] has shown that memory fences and fetch_and_Φ operations are essential in a fundamental way: standard concurrent objects *cannot* be written without them.

CHAPTER 4

Practical Spin Locks

The mutual exclusion problem was first identified in the early 1960s. Dijkstra attributes the first 2-thread solution to Theodorus Dekker [Dijkstra, 1968b]. Dijkstra himself published an n-thread solution in 1965 [CACM]. The problem has been intensely studied ever since. Taubenfeld [2008] provides a summary of significant historical highlights. Ben-Ari [2006, Chaps. 3 & 5] presents a bit more detail. Much more extensive coverage can be found in Taubenfeld's encyclopedic text [Taubenfeld, 2006].

Through the 1960s and '70s, attention focused mainly on algorithms in which the only atomic primitives were assumed to be load and store. Since the 1980s, practical algorithms have all assumed the availability of more powerful atomic primitives, though interest in load/store-only algorithms continues in the theory community.

We present a few of the most important load-store-only spin locks in the first subsection below. In Section 4.2 we consider simple locks based on test_and_set and fetch_and_increment. In Section 4.3 we turn to queue-based locks, which scale significantly better on large machines. Finally, in Section 4.4, we consider additional techniques to reduce unnecessary overhead.

4.1 CLASSICAL LOAD-STORE-ONLY ALGORITHMS

Peterson's Algorithm

The simplest known 2-thread spin lock (Figure 4.1) is due to Peterson [1981]. The lock is represented by a pair of Boolean variables, interested[self] and interested[other] (initially false), and a integer turn that is either 0 or 1. To acquire the lock, thread i indicates its interest by setting interested[self] and then waiting until either (a) the other thread is not interested or (b) turn is set to the other thread, indicating that thread i set it first. As in chapter 3, we use the notation await (*condition*) as shorthand for a spin loop followed by an acquire fence. We do *not* assume that the condition is tested atomically.

To release the lock, thread i sets interested[self] back to false. This allows the other thread, if it is waiting, to enter the critical section. The initial value of turn in each round is immaterial: it serves only to break the tie when both threads are interested in entering the critical section.

In his original paper, Peterson showed how to extend the lock to n threads by proceeding through a series of $n-1$ rounds, each of which eliminates a possible contender. Total (remote-access) time for a thread to enter the critical section, however, is $\Omega(n^2)$, even

```
class lock
    (0, 1) turn
    bool interested[0..1] := { false, false }
lock.acquire():
    interested[self] := true
    fence(WW)
    turn := self
    other := 1 − self
    fence(WR)
    await (interested[other] == false or turn == other)
    fence(RR, RW)
lock.release():
    fence(RW, WW)
    interested[self] := false
```

Figure 4.1: Peterson's 2-thread spin lock. Variable self must be either 0 or 1.

in the absence of contention. In separate work, Peterson and Fischer [1977] showed how to generalize any 2-thread solution to n threads with a hierarchical *tournament* that requires only $O(\log n)$ time, even in the presence of contention. Burns and Lynch [1980] proved that any deadlock-free mutual exclusion algorithm using only reads and writes requires $\Omega(n)$ space.

Defining Time Complexity for Spin Locks

Given that we generally have no bounds on either the length of a critical section or the relative rates of execution of different threads, we cannot in general bound the number of load instructions that a thread may execute in the acquire method of any spin lock algorithm. How then can we compare the time complexity of different locks?

The standard answer is to count only accesses to shared variables (not those that are thread-local), and then only when the access is "remote." This is not a perfect measure, since local accesses are not free, but it captures the dramatic difference in cost between cache hits and misses on modern machines.

On an NRC-NUMA machine, the definition of "remote" is straightforward: we associate a (static) location with each variable and thread, and charge for all and only those accesses made by threads to data at other locations. On a globally cache-coherent machine, the definition is less clear, since whether an access hits in the cache may depend on whether there has been a recent access by another thread. The standard convention is to count all and only those accesses to shared variables that *might* be conflict misses. In a simple loop that spins on a Boolean variable, for example, we would count the initial load that starts the spin and the final load that ends it, but not any loads in-between.

Unless otherwise noted, we will use the globally cache coherent model in this monograph.

```
class lock
    bool choosing[T] := { false ... }
    int number[T] := { 0 ... }
lock.acquire():
    choosing[self] := true
    fence(WR)
    int m := 1 + max_{i∈T} (L→number[i])
    fence(RW)
    number[self] := m
    fence(WW)
    choosing[self] := false
    fence(WR)
    for i ∈ T
        await (choosing[i] == false)
        await (number[i] == 0 or ⟨number[i], i⟩ ≥ ⟨m, self⟩)
    fence(RR, RW)
lock.release():
    fence(RW, WW)
    number[self] := 0
```

Figure 4.2: Lamport's bakery algorithm. The max operation is not assumed to be atomic. It *is*, however, assumed to read each number field only once.

Lamport's Bakery Algorithm

Most of the n-thread mutual exclusion algorithms based on loads and stores can be shown to be starvation free. Given differences in the relative rates of progress of different threads, however, most allow a thread to be bypassed many times before finally entering the critical section. In an attempt to improve fault tolerance, Lamport [1974] proposed a "bakery" algorithm (Figure 4.2) inspired by the "please take a ticket" and "now serving" signs seen at bakeries and other service counters. His algorithm has the arguably more significant advantage that threads acquire the lock in the order in which they first indicate their interest—i.e., in FIFO order.

The second await statement uses lexicographic comparison of ⟨value, thread id⟩ pairs to resolve ties in the numbers chosen by threads. The equals case in the comparison avoids the need for special-case code when a thread examines its own number field. Like all deadlock-free mutual exclusion algorithms based only on loads and stores, the bakery algorithm requires $\Omega(n)$ space. Total time to enter the critical section is also $\Omega(n)$, even in the absence of contention. As originally formulated (and as shown in Figure 4.2), number fields in the bakery algorithm grow without bound. Taubenfeld [2004] has shown how to bound them instead. For machines with more powerful atomic primitives, the conceptually similar *ticket lock* [Fischer et al., 1979; Reed and Kanodia, 1979] (Section 4.2.2) uses

fetch_and_increment on shared "next ticket" and "now serving" variables to reduce space requirements to $O(1)$ and time to $O(m)$, where m is the number of threads concurrently competing for access.

Lamport's Fast Algorithm

One of the key truisms of parallel computing is that if a lock is highly contended most of the time, then the program in which it is embedded probably won't scale. Turned around, this observation suggests that in a well designed program, the typical spin lock will be usually be free when a thread attempts to acquire it. Lamport's "fast" algorithm [1987] (Figure 4.3) exploits this observation by arranging for a lock to be acquired in constant time in the absence of contention (but in $O(n)$ time, where n is the total number of threads in the system, whenever contention is encountered).

The core of the algorithm is a pair of lock fields, x and y. To acquire the lock, thread t must write its id into x and then y, and be sure that no other thread has written to x in-between. Thread t checks y immediately after writing x, and checks x immediately after writing y. If y is not $\perp$ when checked, some other thread must be in the critical section; t waits for it to finish and retries. If x is not still t when checked, some other thread may have entered the critical section (t cannot be sure); in this case, t must wait until the competing thread(s) have either noticed the conflict or left the critical section.

To implement its "notice the conflict" mechanism, the fast lock employs an array of trying flags, one per thread. Each thread sets its flag while competing for the lock (it also leaves it set while executing a critical section for which it encountered no contention). If a thread t *does* detect contention, it unsets its trying flag, waits until the entire array is clear, and then checks to see if it was the *last* thread to set y. If so, it enters the critical section. If not, it retries the acquisition protocol.

A disadvantage of the fast lock as originally presented is that it requires $\Omega(n)$ time (where n is the total number of threads in the system) even when only two threads are competing for the lock. Building on the work of Moir and Anderson [1995], Merritt and Taubenfeld [2000] show how to reduce this time to $O(m)$, where m is the number of threads concurrently competing for access.

4.2 CENTRALIZED ALGORITHMS

As noted in Sections 1.3 and 2.3, almost every modern machine provides read-modify-write (fetch_and_Φ) instructions that can be used to implement mutual exclusion in constant space and—in the absence of contention—constant time. The locks we will consider in this section—all of which use such instructions—differ in fairness and in the performance they provide in the presence of contention.

```
class lock
    𝒯 x
    𝒯 y := ⊥
    bool trying[𝒯] := { false ... }
lock.acquire():
    loop
        trying[self] := true
        fence(WW)
        x := self
        fence(WR)
        if y ≠ ⊥
            trying[self] := false
            fence(WR)
            await (y == ⊥)
            continue            // go back to top of loop
        y := self
        fence(WR)
        if x ≠ self
            trying[self] := false
            fence(WR)
            for i ∈ 𝒯
                await (trying[i] == false)
            fence(RR)
            if y ≠ self
                await (y == ⊥)
                continue        // go back to top of loop
        break
    fence(RR, RW)
lock.release():
    fence(RW, WW)
    y := ⊥
    trying[self] := false
```

Figure 4.3: Lamport's fast algorithm.

4.2.1 TEST_AND_SET LOCKS

The simplest mutex spin lock embeds a test_and_set instruction in a loop, as shown in Figure 4.4. On a typical machine, the test_and_set instruction will require write permission on the target location, necessitating communication across the processor-memory interconnect on every iteration of the loop. These messages will typically serialize, inducing enormous hardware contention that interferes not only with other threads that are attempting to acquire the lock, but also with any attempt by the lock owner to release the lock.

Performance can be improved by arranging to obtain write permission on the lock only when it appears to be free. Proposed by Rudolph and Segall [1984], this *test-and-*test_and_set

```
class lock
    bool f := false
lock.acquire():
    while !TAS(&f);              // spin
    fence(RR, RW)
lock.release():
    fence(RW, WW)
    f := false
```

Figure 4.4: The simple test_and_set lock.

```
class lock
    bool f := false
lock.acquire():
    while !TAS(&f)
        while f;                 // spin
    fence(RR, RW)
lock.release():
    fence(RW, WW)
    f := false
```

Figure 4.5: The test-and-test_and_set lock. Unlike the test_and_set lock of Figure 4.4, this code will typically induce interconnect traffic only when the lock is modified by another core.

lock is still extremely simple (Figure 4.5), and tends to perform well on machines with a small handful of cores. Whenever the lock is released, however, every competing thread will fall out of its inner loop and attempt another test_and_set, each of which induces coherence traffic. With n threads continually attempting to execute a critical sections, total time *per* acquire-release *pair* will be $O(n)$, which is still unacceptable on a machine with more than a handful of cores.

Drawing inspiration from the classic Ethernet contention protocol [Metcalfe and Boggs, 1976], Anderson et al. [1990] proposed an exponential backoff strategy for test_and_set locks (Figure 4.6). Experiments indicate that it works quite well in practice, leading to near-constant overhead per acquire-release pair on many machines. Unfortunately, it depends on constants (the base, multiplier, and limit for backoff) that have no single best value in all situations. Ideally, they should be chosen individually for each machine and workload. Note that test_and_set suffices in the presence of backoff; test-and-test_and_set is not required.

```
class lock
    bool f := false

    const int base = ...
    const int limit = ...           // tuning parameters
    const int multiplier = ...
lock.acquire():
    int delay := base
    while !TAS(&f)
        pause(delay)
        delay := min(delay × multiplier, limit)
    fence(RR, RW)

lock.release():
    fence(RW, WW)
    f := false
```

Figure 4.6: The test_and_set lock with exponential backoff. The pause(k) operation is typically an empty loop that iterates k times. Ideal choices of base, limit and multiplier values depend on the machine architecture and, typically, the application workload.

4.2.2 THE TICKET LOCK

Test_and_set locks are potentially unfair. While most machines can be expected to "randomize" the behavior of test_and_set (e.g., so that some particular core doesn't always win when more than one attempts a test_and_set at roughly the same time), and while exponential backoff can be expected to inject additional variability into the lock's behavior, it is still entirely possible for a thread that has been waiting a very long time to be passed up by a relative newcomer; in principle, a thread can starve.

The *ticket lock* [Fischer et al., 1979; Reed and Kanodia, 1979] (Figure 4.7) addresses this problem. Like Lamport's bakery lock, it grants the lock to competing threads in first-come-first-served order. Unlike the bakery lock, it uses fetch_and_increment to get by with constant space, and with time (per lock acquisition) roughly linear in the number of competing threads.

The code in Figure 4.7 employs a backoff strategy due to Mellor-Crummey and Scott [1991b]. It leverages the fact that my_ticket − L→now_serving represents the number of threads ahead of the calling thread in line. If those threads consume an average of $k\times$ base time per critical section, the calling thread can be expected to probe now_serving about k times before acquiring the lock. Under high contention, this can be substantially smaller than the $O(n)$ probes expected without backoff.

In a system that runs long enough, the next_ticket and now_serving counters can be expected to exceed the capacity of a fixed word size. Rollover is harmless, however:

```
const int base = ...                    // tuning parameter
class lock
    int next_ticket := 0
    int now_serving := 0

lock.acquire():
    int my_ticket := FAI(&next_ticket)
        // returns old value; arithmetic overflow is harmless
    loop
        int ns := now_serving
        if ns == my_ticket
            break
        pause(base × (my_ticket − now_serving))
            // overflow in subtraction is harmless
    fence(RR, RW)

lock.release():
    fence(RW, WW)
    now_serving++
```

Figure 4.7: The ticket lock with proportional backoff. Tuning parameter base should be chosen to be roughly the length of a trivial critical section.

the maximum number of threads in any reasonable system will be less than the largest representable integer, and subtraction works correctly in the ring of integers mod 2^{wordsize}.

4.3 QUEUED SPIN LOCKS

Even with proportional backoff, a thread can perform an arbitrary number of remote accesses in the process of acquiring a ticket lock. Anderson et al. [1990] and (independently) Graunke and Thakkar [1990] showed how to reduce this to a small constant on a globally cache-coherent machine. Both locks employ an array of n flag words, where n is the maximum number of threads in the system. Both arrange for every thread to spin on a different element of the array, and to know the index of the element on which its successor is spinning. In Anderson et al.'s lock, elements are allocated dynamically using fetch_and_increment; a thread releases the lock by updating the next element (in circular order) after the one on which it originally spun. In Graunke and Thakkar's lock, elements are statically allocated. A thread releases the lock by writing its own element; it finds the element on which to spin by performing a swap on an extra tail element.

Inspired by the QOLB hardware primitive of the Wisconsin Multicube [Goodman et al., 1989] and the IEEE SCI standard [Aboulenein et al., 1994] (Section 2.3.2), Mellor-Crummey and Scott [1991b] devised a queue-based spin lock that employs a linked list instead of an array. Craig and, independently, Magnussen, Landin, and Hagersten devised an alternative version that, in essence, links the queue in the opposite direction. Unlike

```
type qnode = record
    qnode* next
    bool locked
class lock
    qnode* tail := null

lock.acquire(qnode* I):
    I→next := null
    I→waiting := true                // Initialization of waiting can be delayed until the if
    fence(WW)                        // statement below, but at the cost of an extra WW fence.
    qnode* prev := swap(&tail, I)
    if prev != null                  // queue was nonempty
        prev→next := I
        while I→locked;              // spin
    fence(RR, RW)

lock.release(qnode* I):
    fence(RW, WW)
    if I→next == null                // no known successor
        if CAS(&tail, I, null) return
        while I→next == null;        // spin
    I→next→locked := false
```

Figure 4.8: The MCS queued lock.

the locks of Anderson et al. and Graunke and Thakkar, these list-based locks require total space $O(n + j)$ for n threads and j locks, rather than $O(nj)$. They are generally considered the methods of choice for FIFO locks on large-scale systems. (Note, however, that strict FIFO ordering may be inadvisable on a system with preemption; see Chapter 7.)

4.3.1 THE MCS LOCK

Pseudocode for the MCS lock appears in Figure 4.8. Every thread using the lock allocates a qnode record containing a queue link and a Boolean flag. Typically, this record lies in the stack frame of the code that calls acquire and release; it must be passed as an argument to both (but see the discussion under "Modifications for a Standard Interface" below).

Threads holding or waiting for the lock are chained together, with the link in the qnode of thread t pointing to the qnode of the thread to which t should pass the lock when done with its critical section. The lock itself is simply a pointer to the qnode of the thread at the tail of the queue, or null if the lock is free.

Operation of the lock is illustrated in Figure 4.9. The acquire method allocates a new qnode, initializes its next pointer to null, and swaps it into the tail of the queue. If the value returned by the swap is null, then the calling thread has acquired the lock (line 2). If the value returned by the swap is non-null, it refers to the qnode of the caller's predecessor in the queue (indicated by the dashed arrow in line 3). Here thread B must set A's next

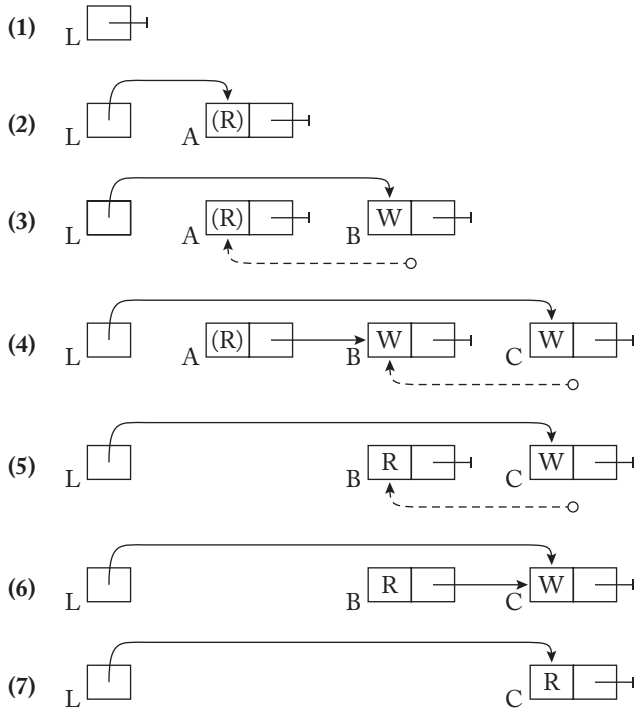

Figure 4.9: Operation of the MCS lock. An 'R' indicates that the thread owning the given qnode is running its critical section (parentheses indicate that the value of the waiting flag is immaterial). A 'W' indicates that the corresponding thread is waiting. A dashed arrow represents a local pointer (returned to the thread by swap).

pointer to refer to its own qnode. Meanwhile, some other thread C may join the queue (line 4).

When thread A has completed its critical section, the release method reads the next pointer of A's qnode to find the qnode of its successor B. It changes B's waiting flag to false, thereby granting it the lock (line 5).

If release finds that the next pointer of its qnode is null, it attempts to CAS the lock tail pointer back to null. If some other thread has already swapped itself into the queue (line 5), the CAS will fail, and release will wait for the next pointer to become non-null (line 6). If there are no waiting threads (line 7), the CASCASpcoderelease will succeed, returning the lock to the appearance in line 1.

The MCS lock has several important properties. Threads join the queue in a wait-free manner (using swap), after which they receive the lock in FIFO order. Each waiting thread spins on a separate location, eliminating contention for cache and interconnect resources.

In fact, because each thread allocates its own qnode, it can arrange for it to be local even on an NRC-NUMA machine. Total (remote access) time to pass the lock from one thread to the next is constant. Total space is linear in the number of threads and locks.

As written (Figure 4.8), the MCS lock requires both swap and compare_and_swap. CAS can of course be used to emulate the swap in the acquire method, but entry to the queue drops from wait-free to lock-free. Mellor-Crummey and Scott [1991b] also show how to make do with only swap in the release method, at the potential cost of FIFO ordering when a thread enters the queue just as its predecessor is releasing the lock.

Modifications for a Standard Interface. One disadvantage of the MCS lock is the need to pass a qnode pointer to acquire and release. Test_and_set and ticket locks pass only a reference to the lock itself. If a programmer wishes to convert code from traditional to queued locks, or to design code in which the lock implementation can be changed at system configuration time, it is natural to wish for a version of the MCS lock that omits the extra parameters. Auslander et al. [2003] devised such a version as part of the K42 project at IBM Research [Appavoo et al., 2005]. Their code exploits the fact that once a thread has acquired a lock, its qnode serves only to hold a reference to the next thread in line. Since the thread now "owns" the lock, it can move its next pointer to an extra field of the lock, at which point the qnode can be discarded.

Code for the K42 variant of the MCS lock appears in Figure 4.10. Operation of the lock is illustrated in Figure 4.11. An idle, unheld lock is represented by a qnode containing two null pointers (line 1 of Figure 4.11). The first of these is the usual tail pointer from the MCS lock; the other is a "next" pointer that will refer to the qnode of the first waiting thread, if and when there is one. Newly arriving thread A (line 2) uses compare_and_swap to replace a null tail pointer with a pointer to the lock variable itself, indicating that the lock is held, but that no other threads are waiting. At this point, newly arriving thread B (line 3) will see the lock variable as its "predecessor," and will update the next field of the lock rather than that of A's qnode (as it would have in a regular MCS lock). When thread C arrives (line 4), it updates B's next pointer, because it obtained a pointer to B's qnode when it performed a CAS on the tail field of the lock. When A completes its critical section (line 5), it finds B's qnode by reading the head field of the lock. It then changes B's "head" pointer (which serves as a waiting flag) to null, thereby releasing B. Upon leaving its spin, B updates the head field of the lock to refer to C's qnode. Assuming no other threads arrive, when C completes its critical section it will return the lock to the state shown in line 1.

The careful reader may notice that the code of Figure 4.10 has a lock-free (not wait-free) entry protocol, and thus admits the (remote, theoretical) possibility of starvation. This can be remedied by replacing the original CAS with a swap, but a thread that finds that the lock was previously free must immediately follow up with a CAS, leading to significantly poorer performance in the (presumably common) uncontended case. A potentially attractive

```
type qnode = record
    qnode* tail
    qnode* next
const qnode* waiting = 1
    // In a real qnode, tail == null means the lock is free;
    // In the qnode that is a lock, tail is the real tail pointer.
class lock
    qnode q := { null, null }

lock.acquire():
    loop
        qnode* prev := q.tail
        if prev == null                          // lock appears to be free
            if CAS(&q.tail, null, &q) break
        else
            qnode I := { waiting, null }
            fence(WW)
            if CAS(&q.tail, prev, &I)            // we're in line
                prev→next := &I
                while I→tail == waiting;         // spin
                // now we have the lock
                qnode* succ := I.next
                if succ == null
                    q.next := null
                    fence(WW)
                    // try to make lock point at itself:
                    if !CAS(&q.tail, &I, &q)
                        // somebody got into the timing window
                        repeat succ := I.next until succ != null
                        q.next := succ
                    break
                else
                    q.next := succ
                    break
    fence(RR, RW)

lock.release():
    fence(RW, WW)
    qnode* succ := q.next
    if succ == null
        if CAS(&q.tail, &q, 0) return
        repeat succ := I.next until succ != null
    succ→tail := null
```

Figure 4.10: K42 variant of the MCS queued lock. Note the standard interface to acquire and release, with no parameters other than the lock itself.

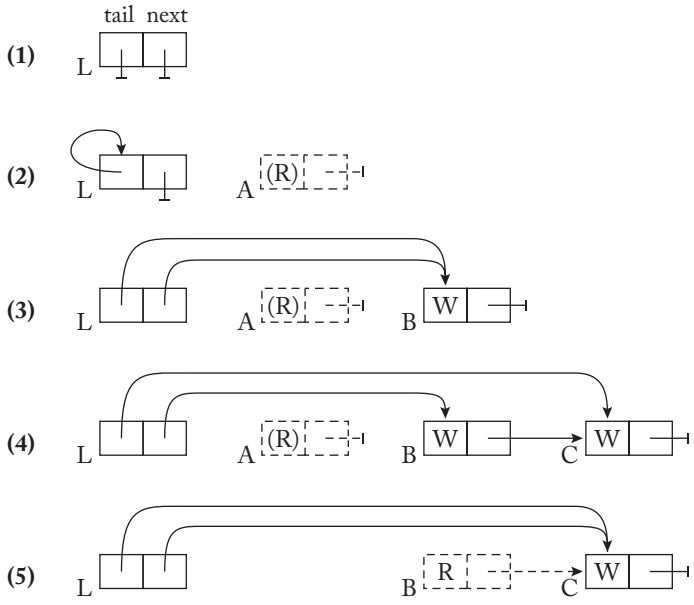

Figure 4.11: Operation of the K42 MCS lock. An 'R' indicates a null "tail" pointer; 'W' indicates non-null. Dashed boxes indicate qnodes that are no longer needed, and may safely be freed by returning from the method in which they were declared.

hybrid strategy starts with a load of the tail pointer, following up with a CAS if the lock appears to be free and a swap otherwise.

4.3.2 THE CLH LOCK

Because every thread spins on a field of its own qnode, the MCS lock achieves a constant bound on the number of remote accesses per lock acquisition, even on a NRC-NUMA machine. The cost of this feature is the need for a newly arriving thread to write the address of its qnode into the qnode of its predecessor, and for the predecessor to wait for that write to complete before it can release a lock whose tail pointer no longer refers to its own qnode.

Craig [1993] and, independently, Magnussen, Landin, and Hagersten [1994] observed that this extra "handshake" can be avoided by arranging for each thread to spin on its predecessor's qnode, rather than its own. On a globally cache-coherent machine, the spin will still be local, because the predecessor's node will migrate to the successor's cache. The downside of the change is that a thread's qnode must potentially remain accessible long after the thread has left its critical section: we cannot bound the time that may elapse

```
type qnode = record
    qnode* prev
    bool succ_must_wait
class lock
    qnode dummy := { null, false }
    // ideally, dummy and tail should lie in separate cache lines
    qnode* tail := &dummy

lock.acquire(qnode* I):
    I→succ_must_wait := true
    fence(WW)
    qnode* pred := I→prev := swap(&tail, I)
    while pred→succ_must_wait;                  // spin
    fence(RR, RW)

lock.release(qnode** I):
    fence(RW, WW)
    qnode* pred := (*I)→prev
    (*I)→succ_must_wait := false
    *I := pred                                  // take pred's qnode
```

Figure 4.12: The CLH queued lock.

before a successor needs to inspect that node. This requirement is accommodated by having a thread provide a fresh qnode to acquire, and return with a *different* qnode from release.

In their original paper, Magnussen, Landin, and Hagersten presented two versions of their lock: a simpler "LH" lock and an enhanced "M" lock; the latter reduces the number of cache misses in the uncontended case by allowing a thread to keep its original qnode when no other thread is trying to acquire the lock. The M lock needs compare_and_swap to resolve the race between a thread that is trying to release a heretofore uncontended lock and the arrival of a new contender. The LH lock has no such races; all it needs is swap.

Craig's lock is essentially identical to the LH lock: it differs only in the mechanism used to pass qnodes to and from the acquire and release methods. It has become conventional to refer to this joint invention by the initials of all three inventors: CLH.

Code for the CLH lock appears in Figure 4.12. An illustration of its operation appears in Figure 4.13. A free lock (line 1 of the latter figure) contains a pointer to a qnode whose succ_must_wait flag is false. Newly arriving thread A (line 2) obtains a pointer to this node (dashed arrow) by executing a swap on the lock tail pointer. It then spins on this node (or simply observes that its succ_must_wait flag is already false). Before returning from acquire, it stores the pointer into its own qnode so it can find it again in release. (In the LH version of the lock [Magnussen, Landin, and Hagersten, 1994], there was no pointer in the qnode; rather, the API for acquire returned a pointer to the predecessor qnode as an explicit parameter.)

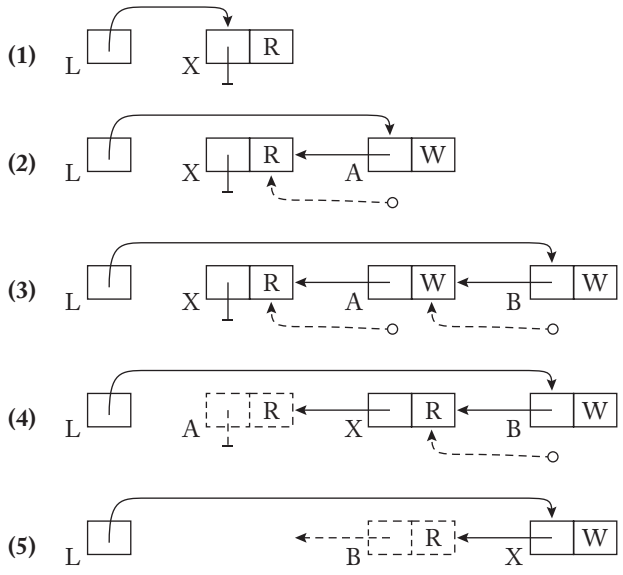

Figure 4.13: Operation of the CLH lock. An 'R' indicates that a thread spinning on this qnode (i.e., the successor of the thread that provided it) is free to run its critical section; a 'W' indicates that it must wait. Dashed boxes indicate qnodes that are no longer needed by successors, and may be reused by the thread releasing the lock. Note the change in label on such nodes, indicating that they now "belong" to a different thread.

To release the lock (line 4), thread A writes false to the succ_must_wait field of its own qnode and then leaves that qnode behind, returning with its predecessor's qnode instead (here previously marked 'X'). Thread B, which arrived at line 3, releases the lock in the same way. If no other thread is waiting at this point, the lock returns to the state in line 1.

In his original paper, Craig [1993] explored several extensions to the CLH lock. By introducing an extra level of indirection, one can eliminate remote spinning even on an NRC-NUMA machine—without requiring compare_and_swap, and without abandoning strict either FIFO ordering or wait-free entry. By linking the list both forward and backward, and traversing it at acquire time, one can arrange to grant the lock in order of some external notion of priority, rather than first-come-first-served (Markatos [1991] presented a similar technique for MCS locks). By marking nodes as abandoned, and skipping over them at release time, one can accommodate time-out (we will consider this topic further in Chapter 7, together with the possibility—suggested by Craig as future work—of skipping over threads that are currently preempted). Finally, Craig sketched a technique to accommodate nested critical sections without requiring a thread to allocate multiple qnodes: arrange for the thread to acquire its predecessor's qnode when the lock is acquired rather than when it

```
qnode initial_thread_qnodes[T]
qnode* thread_qnode_ptrs[T] := { i ∈ T: &initial_thread_qnodes[i] }
type qnode = record
    bool succ_must_wait
class lock
    qnode dummy := { false }
    // ideally, dummy should lie in a separate cache line from tail and head
    qnode* tail := dummy
    qnode* head

lock.acquire():
    qnode* I := thread_qnode_ptrs[self]
    I→succ_must_wait := true
    fence(WW)
    qnode* pred := I→prev := swap(&tail, I)
    while pred→succ_must_wait;                    // spin
    head := I
    thread_qnode_ptrs[self] := pred
    fence(RR, RW)

lock.release():
    fence(RW, WW)
    head→succ_must_wait := false
```

Figure 4.14: A CLH variant with standard interface.

is released, and maintain a separate thread-local stack of pointers to the qnodes that must be modified in order to release the locks.

Modifications for a Standard Interface. Craig's suggestion for nested critical sections requires that locks be released in the reverse of the order in which they were acquired; it does not generalize easily to idioms like hand-over-hand locking (Section 3.1.2). If we adopt the idea of a head pointer field from the K42 MCS lock, however, we can devise a (previously unpublished) CLH variant that serves as a "plug-in" replacement for traditional locks (Figure 4.14).

Our code assumes a global array, thread_qnode_ptrs, indexed by thread id. In practice this could be replaced by any form of "thread-local" storage—e.g., the Posix pthread_getspecific mechanism. Operation is very similar to that of the original CLH lock: the only real difference is that instead of requiring the caller to pass a qnode to release, we leave that pointer in a head field of the lock. No dynamic allocation of qnodes is required: the total number of extant nodes is always $n + j$ for n threads and j locks—one per lock at the head of each queue (with succ_must_wait true or false, as appropriate), one enqueued (not at the head) by each thread currently waiting for a lock, and one (in the appropriate slot of thread_qnode_ptrs) reserved for future use by each thread not currently waiting for a

lock. (Elements of thread_qnode_ptrs corresponding to threads currently waiting for a lock are overwritten [at the end of acquire] before being read again.)

4.3.3 WHICH SPIN LOCK SHOULD I USE?

On modern machines, there is little reason to consider load-store-only spin locks, except perhaps as an optimization in programs with highly asymmetric access patterns (see Section 4.4.4 below).

For small numbers of threads—certainly no more than single digits—both the test_and_set lock with exponential backoff and the ticket lock with proportional backoff tend to work extremely well. The ticket lock is fairer, but this can actually be a disadvantage in some situations (see the discussion of locality-conscious locking in Section 4.4.2 and of scheduling anomalies in Section 7.6).

The problem with both test_and_set and ticket locks is their brittle performance as the number of contending threads increases. In any application in which lock contention may be a bottleneck—even rarely—it makes sense to use a queue-based lock. Here the choice between MCS and CLH locks depends on architectural features and costs. The MCS lock is generally preferred on an NRC-NUMA machine: the CLH lock can be modified to avoid remote spinning, but the extra level of indirection requires additional fetch_and_Φ operations on each lock transfer. For machines with global cache coherence, either lock can be expected to work well. Given the absence of dummy nodes, space needs are lower for CLH locks, but performance may be better (by a small constant factor) on some machines.

4.4 SPECIAL-CASE OPTIMIZATIONS

Many techniques have been proposed to improve the performance of spin locks in important special cases. We will consider the most important of these—read-mostly synchronization—in Chapter 6. In this section we consider three others. Locality-conscious locking biases the acquisition of a lock toward threads that are physically closer to the most recent prior holder, thereby reducing average hand-off cost on NUMA machines. Double-checked locking addresses situations in which initialization of a variable must be synchronized, but subsequent use need not. Asymmetric locking addresses situations in which a lock is accessed repeatedly by the same thread, and performance may improve if that thread is able to reacquire the lock more easily than others can acquire it.

4.4.1 NESTED LOCKS

Throughout this chapter we have been assuming that the access pattern for a given lock is always a sequence of acquire-release pairs, in which the release method is called by the same thread that called acquire—and before the thread attempts to acquire the same lock again.

There are times, however, when it may be desirable to allow a thread to acquire the same lock multiple times, so long as it releases it the same number of times before any other thread acquires it. Suppose, for example, that operation foo accesses data protected by lock L, and that foo is sometimes called by a thread that already holds L, and sometimes by a thread that does not. In the latter case, foo needs to acquire L. With most of the locks presented above, however, the program will deadlock in the former case, where L is already held.

A simple solution, usable with mutual exclusion lock (spin or scheduler-based), is to augment the lock with an owner field and a counter:

```
class nestable_lock
    lock L
    int owner := none
    int count := 0

nestable_lock.acquire()
    if owner != self
        fence(RR)
        L.acquire()
        fence(RW)
        owner := self
    count++
nestable_lock.release()
    if –count == 0
        owner := none
        fence(WW)
        L.release()
```

The astute reader may notice that the read of owner in nestable_lock.acquire races with the writes of owner in both acquire and release. In memory models that forbid data races, the owner field may need to be declared volatile or atomic.

4.4.2 LOCALITY-CONSCIOUS LOCKING

On a NUMA machine—or even one with a non-uniform cache architecture (sometimes known as NUCA)—inter-core communication costs may differ dramatically. If, for example, we have multiple processors, each with multiple cores, we may be able to pass a lock to another core within the same processor much faster than we can pass it to a core of another processor. More significantly, since locks are typically used to protect shared data structures, we can expect the cache lines of the protected structure to migrate to the acquiring core, and this migration will be cheaper if the core is nearby rather than remote.

Radović and Hagersten [2002] were the first to observe the importance of locality in locking, and to suggest passing locks to nearby cores when possible. Their "RH lock," developed for a machine with two NUMA "clusters," is essentially a pair of local test_and_set locks, one of which is initialized to FREE, the other to REMOTE. A thread attempts to

acquire the lock by swapping its id into the local copy. If it gets back FREE or L_FREE (locally free), it has succeeded. If it gets back a thread id, it backs off and tries again. If it gets back REMOTE, it has become the local representative of its cluster, in which case it spins (with a different set of backoff parameters) on the *other* copy of the lock, attempting to CAS it from FREE to REMOTE. To release the lock, a thread usually attempts to CAS it from its own id to FREE. If this fails, a nearby thread must be spinning, in which case the releasing thread stores L_FREE to the lock. Occasionally (subject to a tuning parameter), a releasing thread immediately writes FREE to the lock, allowing it to be grabbed by a remote contender, even if there are nearby ones as well.

While the RH lock could easily be adapted to larger numbers of clusters, space consumption would be linear in the number of such clusters—a property Radović and Hagersten considered undesirable. Their subsequent "hierarchical backoff" (HBO) [Radović and Hagersten, 2003] lock relies on statistics instead. In effect, they implement a test_and_set lock with compare_and_swap, in such a way that the lock variable indicates the cluster in which the lock currently resides. Nearby and remote threads then use different backoff parameters, so that nearby threads are more likely than remote threads to acquire the lock when it is released.

While a test_and_set lock is naturally unfair (and subject to the theoretical possibility of starvation), the RH and HBO locks are likely to be even less fair in practice. Ideally, one would like to be able to explicitly balance fairness against locality. Toward that end, Dice et al. [2012] present a general NUMA-aware design pattern that can be used with (almost) any underlying locks, including (FIFO) queued locks. Their *cohort* locking mechanism employs a global lock that indicates which cluster currently owns the lock, and a local lock for each cluster that indicates the owning thread. The global lock needs to allow release to be called by a different thread from the one that called acquire; the local lock needs to be able to tell, at release time, whether any other local thread is waiting. Apart from these requirements, cohort locking can be used with any known form of lock. Experimental results indicate particularly high throughput (and excellent fairness, subject to locality) using MCS locks at both the global and cluster level.

While the techniques discussed here improve locality only by controlling the order in which threads acquire a lock, it is also possible to control *which threads* perform the operations protected by the lock, and to assign operations that access similar data to the same thread, to minimize cache misses. Such locality-conscious allocation of work can yield major performance benefits in systems that assign fine-grain computational *tasks* to worker threads, as mentioned briefly in Section 5.3.3. It is also a key feature of the *flat combining* we will mention (again briefly) in Section 8.2.

4.4.3 DOUBLE-CHECKED LOCKING

Many applications employ shared variables that must be initialized before they are used for the first time. A canonical example looks like this:

```
foo* p := null
lock L
foo* get_p():
    L.acquire()
        if p == null
            p := new foo()
        foo* rtn := p
    L.release()
    return rtn
```

Unfortunately, this idiom imposes the overhead of acquiring L on every call to get_p. With care, we can use the *double-checked locking* idiom of Figure 4.15 instead. The fences in the figure are critical. In Java, variable p must be declared volatile; in C++, atomic. Without the fences, the initializing thread may set p to point to not-yet-initialized space, or a reading thread may use fields of p that were prefetched before initialization completed. On machines with highly relaxed memory models (e.g., ARM and POWER), the cost of the fences may be comparable to the cost of locking in the original version of the code, making the "optimization" of limited benefit. On machines with a TSO memory model (e.g., the x86 and SPARC), the optimization is much more appealing, since RR, RW, and WW fences are free. Used judiciously, (e.g., in the Linux kernel for x86), double-checked locking can yield significant performance benefits. Even in the hands of experts, however, it has proven to be a significant source of bugs. As a general rule, it is best avoided in application-level code [Bacon et al., 2001].

4.4.4 ASYMMETRIC LOCKING

Many applications contain data structures that are usually—or even always—accessed by a single thread, but are nonetheless protected by locks, either because they are *occasionally* accessed by another thread, or because the programmer is preserving the ability to reuse code in a future parallel context. Several groups have developed locks that can be *biased* toward a particular thread, whose acquire and release operations then proceed much faster than those of other threads. The HotSpot Java Virtual Machine, for example, uses biased locks to accommodate objects that appear to "belong" to a single thread, and to control re-entry to the JVM by threads that have escaped to native code, and may need to synchronize with a garbage collection cycle that began while they were absent [Dice et al., 2001; Russell and Detlefs, 2006].

On a sequentially consistent machine, one might be tempted to avoid (presumably expensive) fetch_and_Φ operations by using a two-thread load-store-only synchronization

```
foo* p := null
lock L
foo* get_p():
    foo *rtn := p
    fence(RR, RW)
    if rtn == null
        L.acquire()
            rtn := p
            if rtn == null                  // double check
                rtn := new foo()
                fence(WW)
                p := rtn
        L.release()
    return rtn
```

Figure 4.15: Double-checked locking.

algorithm (e.g., Dekker's or Peterson's algorithm) to arbitrate between the preferred (bias-holding) thread and some representative of the other threads. Code might look like this:

```
class lock
    Peterson_lock PL
    general_lock GL

lock.acquire():
    if !(preferred_thread)
        GL.acquire()
    PL.acquire()

lock.release():
    PL.release()
    if !(preferred_thread)
        GL.release()
```

The problem, of course, is that load-store-only acquire routines invariably contain some variant of the Dekker store–load sequence—

```
interested[self] := true                        // store
bool potential_conflict := interested[other]    // load
if potential_conflict ...
```

—and this code works correctly on a non-sequentially consistent machine only when augmented with a (presumably also expensive) WR fence between the first and second lines. The cost of the fence has led several researchers [Dice et al., 2001; Russell and Detlefs, 2006; Vasudevan et al., 2010] to propose *asymmetric* Dekker-style synchronization. Applied to Peterson's lock, the solution looks as shown in Figure 4.16.

The key is the handshake operation on the "slow" (non-preferred) path of the lock. This operation must interact with execution on the preferred thread's core in such a way that

1. if the preferred thread set fast_interested before the interaction, then the non-preferred thread is guaranteed to see it afterward.

```
class lock
    bool fast_turn := true
    bool fast_interested := false
    bool slow_interested := false
    general_lock GL

lock.acquire():
    if preferred_thread
        fast_interested := true
        fence(WW)                          // free on TSO machine
        fast_turn := true
        // WR fence intentionally omitted
        await (!slow_interested or !fast_turn)
    else
        GL.acquire()
        slow_interested := true
        fence(WW)
        fast_turn := false
        fence(WR)                          // slow
        handshake()                        // very slow
        await (!fast_interested or fast_turn)
    fence(RR, RW)                          // free on TSO machine

lock.release():
    fence(RW, WW)                          // free on TSO machine
    if preferred_thread
        fast_interested := false
    else
        GL.release()
        slow_interested := false
```

Figure 4.16: An asymmetric lock built around Peterson's algorithm. The handshake operation on the slow path forces a known ordering with respect to the store–load sequence on the fast path.

2. if the preferred thread did *not* set fast_interested before the interaction, then it (the preferred thread) is guaranteed to see slow_interested afterward.

Handshaking can be implemented in any of several ways, including cross-core interrupts, migration to or from the preferred thread's core, forced un-mapping of pages accessed in the critical section, waiting for an interval guaranteed to contain a WR fence (e.g., a scheduling quantum), or explicit communication with a "helper thread" running on the preferred thread's core. Dice et al. [2001] explore many of these options in detail. Because of their cost, they are profitable only in cases where access by non-preferred threads is exceedingly rare. In subsequent work, Dice et al. [2003] observe that handshaking can be

avoided if the underlying hardware provides coherence at a word granularity, but supports atomic writes at subword granularity.

CHAPTER 5

Spin-based Conditions and Barriers

In Chapter 1 we suggested that almost all synchronization serves to achieve either atomicity or condition synchronization. Chapter 4 considered spin-based atomicity. The current chapter considers spin-based condition synchronization—flags and barriers in particular. Chapter 7 will consider scheduler-based alternatives.

5.1 FLAGS

In its simplest form, a flag is Boolean variable, initially false, on which a thread can wait:

```
class flag
    bool f := false

flag.set():
    fence(RW, WW)
    f := true

flag.await():
    while !f;                    // spin
    fence(RR, RW)
```

Methods set and await are presumably called by different threads. Code for set begins with a release fence; await ends with an acquire fence. These reflect the fact that one typically uses set to indicate that previous operations of the calling thread (e.g., initialization of shared data structure) have completed; one typically uses await to ensure that subsequent operations of the calling thread do not begin until the condition holds.

In some algorithms, it may be helpful to have a reset method:

```
flag.reset():
    f := false
    fence(WW)
```

Before calling reset, a thread must ascertain (generally through application-specific means) that no thread is still using the flag for its previous purpose. The WW fence ensures that any subsequent updates (to be announced by a future set) are seen to happen after the reset.

In an obvious generalization of flags, one can arrange to wait on an arbitrary predicate:

```
class predicate
    abstract bool eval()
        // to be extended by users
predicate.await():
    while !eval();                  // spin
    fence(RR, RW)
```

With compiler or preprocessor support, this can become

```
await( condition ):
    while !( condition );           // spin
    fence(RR, RW)
```

This latter form is of course the notation we have used several times in previous chapters. It must be used with care: the absence of an explicit set method means there is no obvious place to hide the release fence that typically proceeds the setting of a Boolean flag. In any program that spins on nontrivial conditions, a thread that changes a variable that may contribute to such a condition may need to precede the change with an explicit fence or, better yet, release of some shared lock. We will return to generalized await statements when we consider conditional critical regions in Section 7.4.1.

5.2 BARRIER ALGORITHMS

Many applications—simulations in particular—proceed through a series of *phases*, each of which is internally parallel, but must complete in its entirety before the next phase can begin. A typical example might look something like this:

```
barrier b
in parallel for i ∈ T
    repeat
        // do i's portion of the work of a phase
        b.cycle()
    until terminating condition
```

The cycle method of barrier b (sometimes called wait, next, or even barrier) forces each thread i to wait until *all* threads have reached that same point in their execution. Calling cycle acccomplishes two things: it announces to other threads that all work prior to the barrier in the current thread has been completed, and it ensures that all work prior to the barrier in *other* threads has been completed before continuing execution in the current thread. To avoid data races, cycle typically begins with a release (RW, WW) fence and ends with an acquire (RR, RW) fence.

The simplest barriers, commonly referred to as *centralized*, employ a small, fixed-size data structure, and consume $\Omega(n)$ time between the arrival of the first thread and the departure of the last. More complex barriers distribute the data structure among the threads, consuming $O(n)$ or $O(n \log n)$ space, but requiring only $\Theta(\log n)$ time.

For any maximum number of threads n, of course, $\log n$ is a constant, and with hardware support it can be a very *small* constant. Some multiprocessors (e.g., the Cray X/XE/Cascade, SGI UV, and IBM Blue Gene series) exploit this observation to provide special constant-time barrier operations (the Blue Gene machines, though, do not have a global address space). With a large number of processors, constant-time hardware barriers can provide a substantial benefit over log-time software barriers.

In effect, barrier hardware performs a global AND operation, setting a flag or asserting a signal once all cores have indicated their arrival. It may also be useful—especially on NRC-NUMA machines, to provide a global OR operation (sometimes known as *Eureka*) that can be used to determine when any *one* of a group of threads has indicated its arrival. Eureka mechanisms are commonly used for parallel search: as soon as one thread has found a desired element (e.g., in its portion of some large data set), the others can stop looking.

The first subsection below presents a particularly elegant formulation of the centralized barrier. The following four subsections present different log-time barriers; a final subsection summarizes their relative advantages.

5.2.1 THE SENSE-REVERSING CENTRALIZED BARRIER

It is tempting to expect a centralized barrier to be easy to write: just initialize a counter to zero, have each thread perform a fetch_and_increment when it arrives, and then spin until the total reaches the number of threads. The tricky part, however, is what to do the second time around. Barriers are meant to be used repeatedly, and without care it is easy to write code in which threads that reach the next barrier "episode" interfere with threads that have not yet gotten around to leaving the previous episode. Several algorithms that suffer from this bug have actually been published.

Perhaps the cleanest solution is to separate the counter from the spin flag, and to "reverse the sense" of that flag in every barrier episode. Code that embodies this technique appears in Figure 5.1. It is adapted from Hensgen et al. [1988]; Almasi and Gottlieb [1989, p. 445] credit similar code to Isaac Dimitrovsky.

The bottleneck of the centralized barrier is the arrival phase: fetch_and_increment operations will serialize, and each can be expected to entail a remote memory access or coherence miss. Departure will also entail $O(n)$ time, but on a globally cache-coherent machine every spinning thread will have its own cached copy of the sense flag, and post-invalidation refills will generally be able to pipeline, for much lower per-access latency.

5.2.2 SOFTWARE COMBINING

It has long been known that a linear sequence of associative (and, ideally, commutative) operations (a "reduction") can be performed tree-style in logarithmic time [Ladner and Fischer, 1980]. For certain read-modify-write operations (notably fetch_and_add), Kruskal et al. [1988] developed reduction-like hardware support as part of the NYU Ultracomputer

```
class barrier
    int count := 0
    const int n := |T|
    bool sense := true
    bool local_sense[T] := { true ... }
barrier.cycle():
    fence(RW, WW)
    bool s := !local_sense[self]
    local_sense[self] := s                  // each thread toggles its own sense
    if FAI(&count) == n−1
        count := 0
        sense := s                          // last thread toggles global sense
    else
        while sense != s;                   // spin
    fence(RR, RW)
```

Figure 5.1: The sense-reversing centralized barrier.

project [Gottlieb et al., 1983]. On a machine with a log-depth interconnection network (in which a message from processor i to memory module j goes through a $O(\log p)$ internal switching nodes on a p-processor machine), near-simultaneous requests to the same location *combine* at the switching nodes. For example, if operations FAA(I, a) and FAA(I, b) landed in the same internal queue at the same point in time, they would be forwarded on as a single FAA(I, a+b) operation. When the result (say s) returned (over the same path), it would be split into two responses—s and either $(s-a)$ or $(s-b)$—and returned to the original requesters.

While hardware combining tends not to appear on modern machines, Yew et al. [1987] observed that similar benefits could be achieved with an explicit tree in software. A shared variable that is expected to be the target of multiple concurrent accesses is represented as a tree of variables, with each node in the tree assigned to a different cache line. Threads are divided into groups, with one group assigned to each leaf of the tree. Each thread updates the state in its leaf. If it discovers that it is the last thread in its group to do so, it continues up the tree and updates its parent to reflect the collective updates to the child. Proceeding in this fashion, late-coming threads eventually propagate updates to the root of the tree.

Using a software combining tree, Tang and Yew [1990] showed how to create a log-time barrier. Writes into one tree are used to determine that all threads have reached the barrier; reads out of a second are used to allow them to continue. Figure 5.2 shows a variant of this combining tree barrier, as modified by Mellor-Crummey and Scott [1991b] to incorporate sense reversal and to replace the fetch_and_Φ instructions of the second combining tree with simple reads (since no real information is returned).

```
type node = record
    const int k                                  // fan-in of this node
    int count := k
    bool sense := false
    node* parent := ...                          // initialized appropriately for tree
class barrier
    bool local_sense[T] := { true ... }
    node* my_leaf[T] := ...                      // pointer to starting node for each thread
    // initialization must create a tree of nodes (each in its own cache line)
    // linked by parent pointers
barrier.cycle():
    fence(RW, WW)
    combining_helper(my_leaf[self], local_sense[self])      // join the barrier
    local_sense[self] := !local_sense[self]                 // for next barrier
    fence(RR, RW)
combining_helper(node* n, bool my_sense):
    if FAD(&n→count) == 1                        // last thread to reach this node
        fence(RW)
        if n→parent != null
            combining_helper(n→parent)
        n→count := n→k                           // prepare for next barrier
        n→sense := !n→sense                      // release waiting threads
    else
        fence(RR)
        while n→sense != my_sense;               // spin
```

Figure 5.2: A software combining tree barrier. FAD is fetch_and_decrement.

Simulations by Yew et al. [1987] show that a software combining tree can significantly decrease contention for reduction variables, and Mellor-Crummey and Scott [1991b] confirm this result for barriers. At the same time, the need to perform (typically expensive) fetch_and_Φ operations at each node of the tree induces substantial constant-time overhead. On an NRC-NUMA machine, most of the spins can also be expected to be remote, leading to potentially unacceptable contention. The barriers of the next three subsections tend to work much better in practice, making combining tree barriers mainly a matter of historical interest. This said, the notion of combining—broadly conceived—has proven useful in the construction of a wide range of concurrent data structures. We will return to the concept in Section 8.2.

5.2.3 THE DISSEMINATION BARRIER

Building on earlier work on barriers [Brooks III, 1986] and information dissemination [Alon et al., 1987; Han and Finkel, 1988], Hensgen et al. [1988] describe a *dissemination barrier*

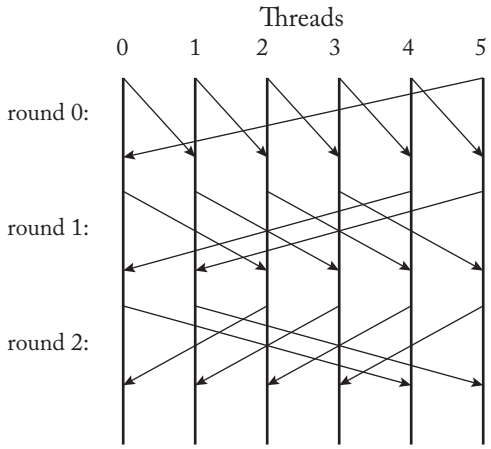

Figure 5.3: Communication pattern for the dissemination barrier (adapted from Hensgen et al. [1988]).

that has no separate arrival and departure phases. The algorithm proceeds through $\lceil \log_2 n \rceil$ (unsynchronized) rounds. In round k, each thread i signals thread $(i + 2^k) \bmod n$. The resulting pattern (Figure 5.3), which works for arbitrary n (not just a power of 2), ensures that by the end of the final round every thread has heard—directly or indirectly—from every other thread.

Code for the dissemination barrier appears in Figure 5.4. The algorithm uses alternating sets of variables (chosen via parity) in consecutive barrier episodes, avoiding interference without requiring two separate spins in each round. It also uses sense reversal to avoid resetting variables after every episode. The flags on which each thread spins are statically determined (allowing them to be local even on an NRC-NUMA machine), and no two threads ever spin on the same flag.

Interestingly, while the critical path length of the dissemination barrier is $\lceil \log_2 n \rceil$, the total amount of interconnect traffic (remote writes) is $n\lceil \log_2 n \rceil$. (Space requirements are also $O(n \log n)$.) This is asymptotically larger than the $O(n)$ space and bandwidth of the centralized and combining tree barriers, and may be a problem on machines whose interconnection network has limited cross-sectional bandwidth.

5.2.4 TOURNAMENT BARRIERS

Tournament barriers [Hensgen et al., 1988; Lubachevsky, 1989] retain the logarithmic critical path and lack of fetch_and_Φ operations found in the dissemination barrier, but reduce communication bandwidth to $O(n)$. Threads begin at the leaves of a binary tree much as they would in a combining tree of fan-in two. One thread from each node continues up the

```
const int logN = ⌈log2 n⌉
type flag_t = record
    bool my_flags[0..1][0..logN−1]
    bool* partner_flags[0..1][0..logN−1]

class barrier
    int parity[T] := { 0 . . . }
    bool sense[T] := { true . . . }
    flag_t flag_array[T] := . . .
        // on an NRC-NUMA machine, flag_array[i] should be local to thread i
        // initially flag_array[i].my_flags[r][k] is false ∀i, r, k
        // if j = (i + 2^k) mod n, then ∀r, k:
        //     flag_array[i].partner_flags[r][k] points to flag_array[j].my_flags[r][k]

barrier.cycle():
    fence(RW, WW)
    flag_t* fp := &flag_array[self]
    int p := parity[self]
    bool s := sense[self]
    for int i in 0..logN−1
        *(fp→partner_flags[p][i]) := s
        fence(WR)
        while fp→my_flags[p][i] != s;      // spin
        fence(RW)
    if p == 1
        sense[self] := !s
    parity[self] := 1 − p
    fence(RR)
```

Figure 5.4: The dissemination barrier.

tree to the next "round" of the tournament. At each stage, however, the "winning" thread is statically determined. In round k (counting from zero) of Hensgen et al.'s barrier, thread i sets a flag awaited by thread j, where $i \equiv 2^k \pmod{2^{k+1}}$ and $j = i - 2^k$. Thread i then drops out of the tournament and busy waits on a global flag for notice that the barrier has been achieved. Thread j participates in the next round of the tournament. A complete tournament consists of $\lceil \log_2 n \rceil$ rounds. Thread 0 sets a global flag when the tournament is over.

Lubachevsky presents one algorithm similar to Hensgen et al.'s, and a second that uses a separate binary tree for departure, similar to that of the combining barrier in Figure 5.2. Because all threads busy wait on a single global flag, Hensgen et al.'s barrier and Lubachevsky's first barrier are appropriate for machines with broadcast-based cache coherence. They will cause heavy interconnect traffic, however, on NRC-NUMA machines, or on machines that limit the degree of cache line replication. Lubachevsky's second barrier could be used on any globally cache coherent machine, including those that use limited-

replication directory-based caching without broadcast. Unfortunately, each thread spins on a non-contiguous set of elements in an array, and no simple scattering of these elements will suffice to eliminate spinning-related network traffic on an NRC-NUMA machine.

Figure 5.5 presents a variant on Hensgen et al.'s tournament barrier, due to Lee [1990] and, independently, Mellor-Crummey and Scott [1991b]. In this variant, each thread spins on its own set of contiguous, statically allocated flags, which are used for both arrival and (tree-based) departure. This layout facilitates local-only spinning (even on an NRC-NUMA machine), but leaves total space consumption at $O(n \log n)$, when $O(n)$ would in principle suffice. In addition to adding a departure tree to Hensgen et al.'s original code, the algorithm uses sense reversal to avoid reinitializing flag variables in each round.

On a machine with broadcast-based global cache coherence, the departure phase of the tournament barrier can profitably be replaced by spinning on a global flag. Experiments by Mellor-Crummey and Scott suggest that the tournament barrier, so modified, will outperform the dissemination barrier on such a machine. On an NRC-NUMA machine with adequate cross-sectional bandwidth, the dissemination barrier is likely to do better, since it achieves the theoretical minimum critical path length.

5.2.5 STATIC TREE BARRIERS

Building on experience with the barriers of the previous three subsections, Mellor-Crummey and Scott [1991b] proposed a static tree barrier that takes logarithmic time and linear space, spins only on local locations (even on an NRC-NUMA machine), and performs the theoretical minimum number of remote memory accesses ($2n - 2$) on machines that lack broadcast.

Code for the barrier appears in Figure 5.7. It incorporates a minor bug fix provided by Kishore Ramachandran. Each thread is assigned a unique tree node which is linked into an arrival tree by a parent link and into a wakeup tree by a set of child links. It is useful to think of the trees as separate because their arity may be different. The code shown here uses an arrival fan-in of 4 and a departure fan-out of 2, which worked well in the authors' original (c. 1990) experiments. Assuming that the hardware supports single-byte writes, fan-in of 4 (on a 32-bit machine) or 8 (on a 64-bit machine) allows a thread to use a single-word spin to wait for all of its arrival-tree children simultaneously. Optimal departure fan-out is likely to be machine-dependent. In principle, it might even make sense to use wider fan-out near the root of the departure tree, but this would lead to considerable code complexity. As in the tournament barrier, wakeup on a machine with broadcast-based global cache coherence could profitably be effected with a single global flag.

5.2.6 WHICH BARRIER SHOULD I USE?

Experience suggests that the centralized, dissemination, and static tree barriers are all useful in certain environments. The centralized barrier has the advantage of simplicity,

```
const int logN = ⌈log2 n⌉
type role_t = (winner, loser, bye, champion, dropout)

type round_t = record
    role_t role
    bool* opponent
    bool flag := false

class barrier
    bool sense[T] := { true ... }
    round_t rounds[T][0..logN] := ...      // see Figure 5.6
        // on an NRC-NUMA machine, rounds[i] should be local to thread i

barrier.cycle():
    fence(RW, WW)
    int round := 1
    bool my_sense := sense[self]
    loop    // arrival
        case rounds[self][round].role of
            loser:
                fence(RR, WR)
                *rounds[self][round].opponent := my_sense
                fence(WR)
                while rounds[self][round].flag != my_sense;      // spin
                break loop
            winner:
                fence(RR, WR)
                while rounds[self][round].flag != my_sense;      // spin
            bye:                // do nothing
            champion:
                fence(RR, WR)
                while rounds[self][round].flag != my_sense;      // spin
                fence(RW)
                *rounds[self][round].opponent := my_sense
                break loop
            dropout:            // impossible
        ++round
    loop    // departure
        --round
        case rounds[self][round].role of
            loser, champion:    // impossible
            winner:             *rounds[self][round].opponent := my_sense
            bye:                // do nothing
            dropout:            break loop
    sense[self] := !my_sense
    fence(RR, RW)
```

Figure 5.5: A tournament barrier with local-spinning tree-based departure. Fences have been made overly conservative for ease of presentation.

rounds[i][k].role =
- winner — if $k > 0$, $i \bmod 2^k = 0$, $i + 2^{k-1} < n$, and $2^k < n$
- bye — if $k > 0$, $i \bmod 2^k = 0$, and $i + 2^{k-1} \geq n$
- loser — if $k > 0$ and $i \bmod 2^k = 2^{k-1}$
- champion — if $k > 0$, $i = 0$, and $2^k \geq n$
- dropout — if $k = 0$
- unused otherwise; value immaterial

rounds[i][k].opponent points to
- rounds[$i - 2^{k-1}$][k].flag if rounds[i][k].role = loser
- rounds[$i + 2^{k-1}$][k].flag if rounds[i][k].role = winner or champion
- unused otherwise; value immaterial

Figure 5.6: Initialization of rounds array for the tournament barrier.

and tends to outperform all other alternatives when the number of threads is small. It also adapts easily to different numbers of threads. In an application in which the number changes from one barrier episode to another, this advantage may be compelling.

Given the cost of remote spinning (and of fetch_and_Φ operations on most machines), the combining tree barrier tends not to be competitive. The tournament barrier, likewise, has little to recommend it over the static tree barrier.

The choice between the dissemination and static tree barriers comes down to a question of architectural features and costs. The dissemination barrier has the shortest critical path, but induces asymptotically more total network traffic. (It is also ill suited to applications that can exploit the "fuzzy" barriers of Section 5.3.1 below.) Given broadcast-based cache coherence, nothing is likely to outperform the static tree barrier, modified to use a global departure flag. In the absence of broadcast, the dissemination barrier will do better on a machine with high cross-sectional bandwidth; otherwise the static tree barrier (with explicit departure tree) is likely to do better. When in doubt, practitioners would be wise to try both and measure their performance.

5.3 BARRIER EXTENSIONS

5.3.1 FUZZY BARRIERS

One of the principal performance problems associated with barriers is *skew* in thread arrival times, often caused by irregularities in the amount of work performed between barrier episodes. If one thread always does more work than the others, of course, then it will always be delayed, and all of the others will wait. If variations are more normally distributed, however, then we arrive at the unpleasant situation illustrated on the left side of Figure 5.8, where the time between barrier episodes is always determined by the slowest thread. If $T_{i,p}$

```
type node = record
    bool parent_sense := false
    bool* parent_ptr
    bool have_child[0..3]                    // for arrival
    bool child_not_ready[0..3]
    bool* child_ptrs[0..1]                   // for departure
    bool dummy                               // pseudodata
class barrier
    bool sense[T] := true
    node nodes[T]
    // on an NRC-NUMA machine, nodes[i] should be local to thread i
    // in nodes[i]:
    //      have_child[j] = true iff 4i + j + 1 < n
    //      parent_ptr = &nodes[⌊(i − 1)/4⌋].child_not_ready[(i − 1) mod 4],
    //           or &dummy if i = 0
    //      child_ptrs[0] = &nodes[2i + 1].parent_sense, or &dummy if 2i + 1 ≥ n
    //      child_ptrs[1] = &nodes[2i + 2].parent_sense, or &dummy if 2i + 2 ≥ n
    //      initially child_not_ready := have_child
barrier.cycle():
    fence(RW, WW)
    node* n := &nodes[self]
    bool my_sense := sense[self]
    while n→child_not_ready != {false, false, false, false};      // spin
    n→child_not_ready := n→have_child        // prepare for next episode
    fence(RW)
    *n→parent_ptr := false                   // let parent know we're ready
    // if not root, wait until parent signals departure:
    if self != 0
        fence(WR)
        while n→parent_sense != my_sense;    // spin
        fence(RW)
    // signal children in departure tree
    *n→child_ptrs[0] := my_sense
    *n→child_ptrs[1] := my_sense
    sense[self] := !my_sense
    fence(RR)
```

Figure 5.7: A static tree barrier with local-spinning tree-based departure.

is the time thread i consumes in phase p of the computation, then total execution time is $\sum_p (t_b + \max_{i \in \mathcal{T}} T_{i,p})$, where t_b is the time required by a single barrier episode.

Fortunately, it often turns out that the work performed in one algorithmic phase depends on only *some* of the work performed by peers in previous phases. If we can arrange for peers to do this critical work first, then we can start the next phase in fast threads as soon as slow threads have finished their critical work—and before they have finished *all*

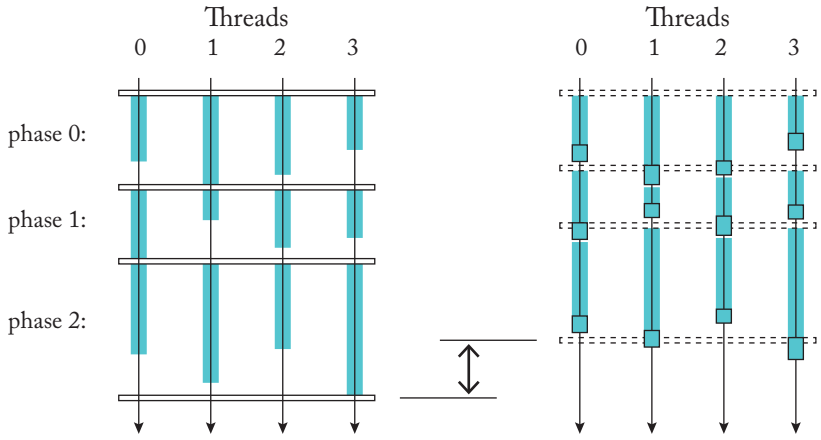

Figure 5.8: Impact of variation across threads in phase execution times, with normal barriers (left) and fuzzy barriers (right). Blue work bars are the same length in each version of the figure. Fuzzy intervals are shown as outlined boxes. With fuzzy barriers, threads can leave the barrier as soon as the last peer has entered its fuzzy interval. Overall performance improvement is shown by the double-headed arrow at center.

their work. This observation, due to Gupta [1989], leads to the design of a *fuzzy* barrier, in which arrival and departure are separate operations. The standard idiom

```
in parallel for i ∈ T
    repeat
        // do i's portion of the work of a phase
        b.cycle()
    until terminating condition
```

becomes

```
in parallel for i ∈ T
    repeat
        // do i's critical work for this phase
        b.arrive()
        // do i's non-critical work—its fuzzy interval
        b.depart()
    until terminating condition
```

As illustrated on the right side of Figure 5.8, the impact on overall run time can be a dramatic improvement.

A centralized barrier is easily modified to produce a fuzzy variant (Figure 5.9). Unfortunately, none of the logarithmic barriers we have considered has such an obvious fuzzy version. We address this issue in the following subsection.

```
class barrier
    int count := 0
    const int n := |T|
    bool sense := true
    bool local_sense[T] := { true ... }
barrier.arrive():
    fence(RW, WW)
    local_sense[self] := !local_sense[self]   // each thread toggles its own sense
    if FAI(&count) == n−1
        count := 0
        sense := local_sense[self]            // last thread toggles global sense
barrier.depart():
    while sense != local_sense[self];         // spin
    fence(RR, RW)
```

Figure 5.9: Fuzzy variant of the sense-reversing centralized barrier.

5.3.2 ADAPTIVE BARRIERS

When all threads arrive at about the same time, the tree, dissemination, and tournament barriers enjoy an asymptotic advantage over the centralized barrier. The latter, however, has an important advantage when thread arrivals are heavily skewed: if all threads but one have already finished their arrival work, the last thread is able recognize this fact in constant time in a centralized barrier. In the other barriers it will almost always require logarithmic time (an exception being the lucky case in which the last-arriving thread just happens to own the root node of the static tree barrier).

This inability to amortize arrival time is the same reason why most log-time barriers do not easily support fuzzy-style separation of their arrival and departure operations. In any fuzzy barrier it is essential that threads wait *only* in the departure operation, and then only for threads that have yet to reach the arrival operation. In the dissemination barrier, no thread knows that all other threads have arrived until the very end of the algorithm. In the tournament and static tree barriers, static synchronization orderings force some threads to wait for their peers before announcing that they have reached the barrier. In all of the algorithms with tree-based departure, threads waiting near the leaves cannot discover that the barrier has been achieved until threads higher in the tree have already noticed this fact.

Among our log-time barriers, only the combining tree appears to offer a way to separate arrival and departure. Each arriving thread works its way up the tree to the topmost unsaturated node (the first at which it was not the last to arrive). If we call this initial traversal the arrive operation, it is easy to see that it involves no spinning. On a machine with broadcast-based global cache coherence, the thread that saturates the root of the tree can flip a sense-reversing flag, on which all threads can spin in their depart operation. On

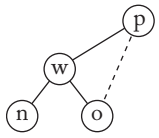

Figure 5.10: Dynamic modification of the arrival tree in an adaptive combining tree barrier.

other machines, we can traverse the tree again in depart, but in a slightly different way: this time a thread proceeds upward only if it is the *first* to arrive at a node, rather than the last. Other threads spin in the top-most node in which they were the first to arrive. The thread that reaches the root waits, if necessary, for the last arriving thread, and then flips flags in each of the children of the root. Any thread that finds a flipped flag on its way up the tree knows that it can stop and flip flags in the children of *that* node. Races among threads take a bit of care to get right, but the code can be made to work. Unfortunately, the last thread to call the arrive operation still requires $\Omega(\log n)$ time to realize that the barrier has been achieved.

To address this remaining issue, Gupta and Hill [1989] proposed an *adaptive combining tree*, in which early-arriving threads dynamically modify the structure of the tree so that late-arriving peers are closer to the root. With modest skew in arrival times, the last-arriving thread realizes that the barrier has been achieved in constant time.

The code for this barrier is somewhat complex. The basic idea is illustrated in Figure 5.10. It uses a binary tree. Each thread, in its arrive operation, starts at its (statically assigned) leaf and proceeds upward, stopping at the first node (say, w) that has not yet been visited by any other thread. It then modifies the tree so that w's other child (o, the child through which the thread did not climb) is one level closer to the root. Specifically, the thread changes o's parent to be p (the parent of w) and makes o a child of p. A thread that reaches p through w's sibling (not shown) will promote o another level, and a later-arriving thread, climbing through o, will traverse fewer levels of the tree than it would have otherwise.

In their paper, Gupta and Hill [1989] present both standard and fuzzy versions of their adaptive combining tree barrier. Unfortunately, both versions retain the remote spins of the original (non-adaptive) combining tree. They also employ test_and_set locks to arbitrate access to each tree node. To improve performance—particularly but not exclusively on NRC NUMA machines—Scott and Mellor-Crummey [1994] present versions of the adaptive combining tree barrier (both regular and fuzzy) that spin only on local locations and that adapt the tree in a wait-free fashion, without the need for per-node locks. In the process they also fix several subtle bugs in the earlier algorithms. Readers interested in exploring adaptive barriers should use these later versions.

5.3.3 BARRIER-LIKE CONSTRUCTS

While barriers are the most common form of global (all-thread) synchronization, they are far from the only one. We have already mentioned the "Eureka" operation in Sections 2.3.2 and 5.2. Invoked by a thread that has discoverd some desired result (hence the name), it serves to interrupt the thread's peers, allowing them (in the usual case) to stop looking for similar results. Whether supported in hardware or software, the principal challenge for Eureka is to cleanly terminate the peers. The easiest solution is to require each thread to poll for termination periodically, but this can be both awkward and wasteful. More asynchronous solutions require careful integration with the thread library or language run-time system, and are beyond the scope of this monograph.

Many languages (and, more awkwardly, library packages) allow the programmer to launch a group of threads and wait for their completion. In Cilk [Frigo et al., 1998], for example, a multi-phase application might look something like this:

```
do {
    for (i = 0; i < n; i++) {
        spawn work(i);
    }
    sync;   // wait for all children to complete
} while (!terminating condition)
```

Semantically, this code suggests the creation of n threads at the top of each do loop iteration, and a "join" among them at the bottom. The Cilk runtime system, however, is designed to make spawn and sync as inexpensive as possible. In effect, sync functions as a barrier among preexisting worker threads, and spawn simply identifies units of work ("*tasks*") that can be farmed out to a worker.

Many languages (including the more recent Cilk++) include a "parallel for" loop whose iterations proceed logically in parallel. An implicit sync causes execution of the main program to wait for all iterations to complete before proceeding with whatever comes after the loop. Like threads executing the same phase of a barrier-based application, iterations of a parallel loop must generally be free of data races. If occasional conflicts are allowed, they must be resolved using other synchronization.

In a very different vein, Fortran 95 and its descendants provide a forall loop whose iterations are heavily synchronized. Code like the following

```
forall (i=1:n)
    A[i] = expr1
    B[i] = expr2
    C[i] = expr3
end forall
```

contains (from a semantic perspective) a host of implicit barriers: All instances of *expr*1 are evaluated first, then all writes are performed to A, then all instances of *expr*2 are

evaluated, followed by all writes to `B`, and so forth. A good compiler will elide any barriers it can prove to be unneeded.

Recognizing the host of different patterns in which parallel threads may synchronize, Shirako et al. [2008] have developed a barrier generalization known as *phasers*. Threads can join (*register with*) or leave a phaser dynamically, and can participate as *signalers*, *waiters*, or both. Their signal and wait operations can be separated by other code to effect a fuzzy barrier. Threads can also, as a group, specify a statement to be executed, atomically, as part of a phaser episode. Finally, and perhaps most importantly, a thread that is registered with multiple phasers can signal or wait at all of them together when it performs a signal or wait operation. This capability facilitates the management of *stencil* applications, in which a thread synchronizes with its neighbors at the end of each phase, but not with other threads. Neighbor-only synchronization is also supported, in a more limited fashion, by the *topological barriers* of Scott and Michael [1996]. In the message-passing word, barrier-like operations are supported by the *collective communication* primitives of systems like MPI [Bruck et al., 1995], but these are beyond the scope of this monograph.

CHAPTER 6

Read-mostly Atomicity

In Chapter 4 we considered the topic of busy-wait mutual exclusion, which achieves atomicity by allowing only one thread at a time to execute a critical section. While mutual exclusion is sufficient to ensure atomicity, it is by no means necessary. Any mechanism that satisfies the ordering constraints of Section 3.1.2 will also suffice. In particular, *read-mostly* optimizations exploit the fact that operations can safely execute concurrently, while still maintaining atomicity, if they read shared data without writing it.

Section 6.1 considers the simplest read-mostly optimization: the reader-writer lock, which allows multiple readers to occupy their critical section concurrently, but requires writers (that is, threads that may update shared data, in addition to reading it) to exclude both readers and other writers. To use the "reader path" of a reader-writer lock, a thread must know at the beginning of the critical section, that it will never attempt to write. Sequence locks, the subject of Section 6.2, relax this restriction by allowing a reader to "upgrade" to writer status if it forces all concurrent readers to back out and retry their critical sections. (Transactional memory, which we will consider in Chapter 9, can be considered a generalization of sequence locks. TM systems typically automate the back-out-and-retry mechanism; sequence locks require the programmer to implement it by hand.) Finally read-copy update (RCU), the subject of Section 6.3 explores an extreme position in which the overhead of synchronization is shifted almost entirely off of readers and onto writers, which are assumed to be quite rare.

6.1 READER-WRITER LOCKS

Reader-writer locks, first suggested by Courtois et al. [1971], relax the constraints of mutual exclusion to permit more than one thread to inspect a shared data structure simultaneously, so long as none of them modifies it. Critical sections are separated into two classes: *writes*, which require exclusive access while modifying protected data, and *reads*, which can be concurrent with one another (though not with writes) because they are known in advance to make no observable changes.

As recognized by Courtois et al. [1971], different fairness properties are appropriate for a reader-writer lock depending on the context in which it is used. A "reader preference" lock minimizes the delay for readers and maximizes total throughput by allowing a reading thread to join a group of current readers even if a writer is waiting. A "writer preference" lock ensures that updates are seen as soon as possible by requiring readers to wait for

any current or waiting writer, even if other threads are currently reading. Both of these options permit indefinite postponement and even starvation of non-preferred threads when competition for the lock is high. Though not explicitly recognized by Courtois et al. [1971], it is also possible to construct a reader-writer lock (called a "fair" lock below) in which readers wait for any earlier writer and writers wait for any earlier thread of either kind.

The locks of Courtois et al. [1971] were based on semaphores, a scheduler-based synchronization mechanism that we will introduce in Section 7.2. In the current chapter we limit ourselves to busy-wait synchronization.

6.1.1 CENTRALIZED ALGORITHMS

There are many ways to construct a centralized reader-writer lock. We consider three examples here.

Our first example (Figure 6.1) gives preference to readers. It uses an unsigned integer to represent the state of the lock. The lowest bit indicates whether a writer is active; the upper bits contain a count of active or interested readers. When a reader arrives, it increments the reader count (atomically) and waits until there are no active writers. When a writer arrives, it attempts to acquire the lock using compare_and_swap. The writer succeeds, and proceeds, only when all bits were clear, indicating that no other writer was active and that no readers were active or interested. Since a reader waits only when a writer is active, and is able to proceed as soon as that one writer finishes, exponential backoff for readers is probably not needed (constant backoff may sometimes be appropriate; we do not consider it here). Since writers may be delayed during the execution of an arbitrary number of critical sections, they use exponential backoff to minimize contention.

The symmetric case—writer preference—appears in Figure 6.2. In this case we must count both active readers (to know when all of them have finished) and interested writers (to know whether a newly arriving reader must wait). We also need to know whether a writer is currently active. Even on a 32-bit machine, a single word still suffices to hold both counts and a Boolean flag. A reader waits until there are no active or waiting writers; a writer waits until there are no active readers. Because writers are unordered (in this particular lock), they use exponential backoff to minimize contention. Readers, on the other hand, can use ticket-style proportional backoff to defer to all waiting writers.

Our final centralized example—a fair reader-writer lock—appears in Figure 6.3. It is patterned after the ticket lock (Section 4.2.2), and is represented by two pairs of counters. Each pair occupies a single word: the upper half of each counts readers; the lower half counts writers. The counters of the request word indicate how many threads have requested the lock. The counters of the completion word indicate how many have already acquired and released it. With arithmetic performed modulo the precision of half-word quantities (and with this number assumed to be significantly larger than the total number of threads), overflow is harmless. Readers spin until all earlier write requests have completed. Writers

```
class lock
    int n := 0
    // low-order bit indicates whether a writer is active;
    // remaining bits are a count of active or waiting readers
    const int WA_flag = 1
    const int RC_inc = 2
    const int base, limit, multiplier = ...        // tuning parameters

lock.reader_acquire():
    (void) FAA(&n, RC_inc)
    fence(WR)
    while n & WA_flag == 1;          // spin
    fence(RR)

lock.reader_release():
    fence(RW)
    (void) FAA(&n, −RC_inc)

lock.writer_acquire():
    int delay := base
    while !CAS(&n, 0, WA_flag)       // spin
        pause(delay)
        delay := min(delay × multiplier, limit)
    fence(RR, RW)

lock.writer_release():
    fence(RW, WW)
    (void) FAA(&n, −WA_flag)
```

Figure 6.1: A centralized reader-preference reader-writer lock, with exponential backoff for writers.

spin until all earlier read and write requests have completed. For both readers and writers we use the difference between requested and completed writer critical sections to estimate the expected wait time. Depending on how many (multi-)reader episodes are interleaved with these, this estimate may be off by as much as a factor of 2.

6.1.2 QUEUED READER-WRITER LOCKS

Just as centralized mutual exclusion locks—even with backoff—can induce unacceptable contention under heavy use on large machines, so too can centralized reader-writer locks. To avoid the contention problem, Mellor-Crummey and Scott [1991a] have shown how to adapt queued spin locks to the reader-writer case. Specifically, they present reader-preference, writer-preference, and fair reader-writer locks based on the MCS lock (Section 4.3.1). We present the fair version in Figures 6.4 and 6.5 (including a bug fix provided by Keir Fraser). For the other two, interested readers may consult the original paper.

```
class lock
    ⟨short, short, bool⟩ n := ⟨0, 0, false⟩
    // high half of word counts active readers; low half counts waiting writers,
    // except for low bit, which indicates whether a writer is active
    const int base, limit, multiplier = ...        // tuning parameters

lock.reader_acquire():
    loop
        ⟨short ar, short ww, bool aw⟩ := n
        if ww == 0 and aw == false
            if CAS(&n, ⟨ar, 0, false⟩, ⟨ar+1, 0, false⟩) break
        // else spin
        pause(ww × base)                           // proportional backoff
    fence(RR)

lock.reader_release():
    fence(RW)
    short ar, ww; bool aw
    repeat    // fetch-and-phi
        ⟨ar, ww, aw⟩ := n
    until CAS(&n, ⟨ar, ww, aw⟩, ⟨ar−1, ww, aw⟩)

lock.writer_acquire():
    int delay := base
    loop
        ⟨short ar, short ww, bool aw⟩ := n
        if aw == false
            if CAS(&n, ⟨ar, ww, false⟩, ⟨ar, ww, true⟩) break
        if CAS(&n, ⟨ar, ww, aw⟩, ⟨ar, ww+1, aw⟩)
            // I'm registered as waiting
            loop                                   // spin
                ⟨ar, ww, aw⟩ := n
                if aw == false
                    if CAS(&n, ⟨ar, ww, false⟩, ⟨ar, ww−1, true⟩) break outer loop
                pause(delay)                       // exponential backoff
                delay := min(delay × multiplier, limit)
        // else retry
    fence(RR, RW)

lock.writer_release():
    fence(RW, WW)
    short rr, wr; bool aw
    repeat    // fetch-and-phi
        ⟨ar, ww, aw⟩ := n
    until CAS(&n, ⟨ar, ww, aw⟩, ⟨ar, ww, false⟩)
```

Figure 6.2: A centralized writer-preference reader-writer lock, with proportional backoff for readers and exponential backoff for writers.

```
class lock
    ⟨short, short⟩ requests := ⟨0, 0⟩
    ⟨short, short⟩ completions := ⟨0, 0⟩
    // top half of each word counts readers; bottom half counts writers
    const int base = ...                    // tuning parameter

lock.reader_acquire():
    short rr, wr, rc, wc
    repeat    // fetch-and-phi
        ⟨rr, wr⟩ := requests
    until CAS(&requests, ⟨rr, wr⟩, ⟨rr+1, wr⟩)
    loop      // spin
        ⟨rc, wc⟩ := completions
        if wc == wr break
        pause((wc−wr) × base)
    fence(RR)

lock.reader_release():
    fence(RW)
    short rc, wc
    repeat    // fetch-and-phi
        ⟨rc, wc⟩ := completions
    until CAS(&completions, ⟨rc, wc⟩, ⟨rc+1, wc⟩)

lock.writer_acquire():
    short rr, wr, rc, wc
    repeat    // fetch-and-phi
        ⟨rr, wr⟩ := requests
    until CAS(&requests, ⟨rr, wr⟩, ⟨rr, wr+1⟩)
    loop      // spin
        ⟨rc, wc⟩ := completions
        if rc == rr and wc == wr break
        pause((wc−wr) × base)
    fence(RR, RW)

lock.writer_release():
    fence(RW, WW)
    short rc, wc
    repeat    // fetch-and-phi
        ⟨rc, wc⟩ := completions
    until CAS(&completions, ⟨rc, wc⟩, ⟨rc+1, wc⟩)
```

Figure 6.3: A centralized fair reader-writer lock with (roughly) proportional backoff for both readers and writers. Addition is assumed to be modulo the precision of (unsigned) short integers.

```
type role_t = (reader, writer, none)
type qnode = record
    role_t role
    qnode* next
    ⟨ role_t successor_role        // these two fields share a word
      bool blocked           ⟩ state
class lock
    qnode* tail := null
    int reader_count := 0
    qnode* next_writer := null

lock.reader_acquire(qnode* I):
    I→role := reader; I→next := null
    I→⟨successor_role, blocked⟩ := ⟨none, true⟩
    fence(WW)
    qnode* pred := swap(&tail, I)
    if pred == null
        (null) FAI(&reader_count)
        I→blocked := false
    else
        if pred→role == writer or else CAS(&pred→state, ⟨none, true⟩, ⟨reader, true⟩)
            // pred is a writer or waiting reader;
            // will inc. reader_count and release me when appropriate
            pred→next := I
            fence(WR)
            while I→blocked;              // spin for permission to proceed
        else
            (void) FAI(&reader_count)
            fence(WW)
            pred→next := I
            I→blocked := false
    // I can go now
    if I→successor_role == reader
        while I→next == null;             // spin for successor identity
        fence(RW)
        (void) FAI(&reader_count)
        fence(WW)
        I→next→blocked := false
    fence(RR)
```

Figure 6.4: A fair queued reader-writer lock (declarations and reader_acquire).

```
lock.reader_release(qnode* I):
    fence(RW)
    if I→next != null or else !CAS(&tail, I, null)
        while I→next == null;              // spin for successor identity
        if I→successor_role == writer
            next_writer := I→next
    if FAD(&reader_count) == 1 and then (qnode* w := next_writer) != null
            and then reader_count == 0 and then CAS(&next_writer, w, null)
        // I'm the last active reader and there exists a waiting writer and there were still
        // no active readers after I became the unique reader to identify the waiting writer
        w→blocked := false
lock.writer_acquire(qnode* I):
    I→role := writer; I→next := null
    I→⟨successor_role, blocked⟩ := ⟨none, true⟩
    fence(WW)
    qnode* pred := swap(&tail, I)
    if pred == null
        next_writer := I
        fence(WR)
        if reader_count == 0 and swap(&next_writer, null) == I
            // no reader who will resume me
            I→blocked := false
    else  // must update successor_role before updating next
        pred→successor_role := writer
        fence(WW)
        pred→next := I
        fence(WR)
    while I→blocked;                       // spin
    fence(RR, RW)
lock.writer_release(qnode* I):
    fence(RW, WW)
    if I→next != null or else not CAS(&tail, I, null)
        while I→next == null;              // spin for successor identity
        if I→next→role == reader
            (void) FAI(&reader_count)
        I→next→blocked := false
```

Figure 6.5: A fair queued reader-writer lock (continued).

As in the MCS spin lock, we use a linked list to keep track of requesting threads, and we pass a qnode pointer to both the acquire and release routines. (In the typical case, the qnode will be allocated in the stack frame of the routine that calls both acquire and release.) In contrast to the original MCS lock, however, we allow a requestor to read and write fields in the qnode of its predecessor (if any). To ensure that the node is still valid (and has not been deallocated or reused), we require that a thread access its predecessor's

qnode before initializing the node's next pointer. At the same time, we force every thread that has a successor to wait for its next pointer to become non-null, even if the pointer will never be used. As in the MCS lock, the existence of a successor is determined by examining L→tail.

A reader can begin reading if its predecessor is a reader that is already active, but it must first unblock its successor (if any) if that successor is a waiting reader. To ensure that a reader is never left blocked while its predecessor is reading, each reader uses compare_and_swap to atomically test if its predecessor is an active reader, and if not, notify its predecessor that it has a waiting reader as a successor.

Similarly, a writer can proceed if its predecessor is done and there are no active readers. A writer whose predecessor is a writer can proceed as soon as its predecessor is done, as in the MCS lock. A writer whose predecessor is a reader must go through an additional protocol using a count of active readers, since some readers that started earlier may still be active. When the last reader of a group finishes (reader_count == 0), it must resume the writer (if any) next in line for access. This may require a reader to resume a writer that is not its direct successor. When a writer is next in line for access, we write its name in a global location. We use swap to read and erase this location atomically, ensuring that a writer proceeds on its own if and only if no reader is going to try to resume it. To make sure that reader_count never reaches zero prematurely, we increment it before resuming a blocked reader, and before updating the next pointer of a reader whose reading successor proceeds on its own.

6.2 SEQUENCE LOCKS

For read-mostly workloads, reader-writer locks still suffer from two significant limitations. First, a reader must know that it *is* a reader, before it begins its work. A deeply nested conditional that occasionally—but very rarely—needs to modify shared data will force the surrounding critical section to function as a writer every time. Second, a reader must write the metadata of the lock itself to ward off simultaneous writers. Because the write requires exclusive access, it is likely to be a cache miss when multiple readers are active. Given the cost of a miss, lock overhead can easily dominate the cost of other operations in the critical section.

Sequence locks (*seqlocks*) [Lameter, 2005] address these limitations. A reader is allowed to "change its mind" and become a writer in the middle of a critical section. More significantly, readers only read the lock—they do not update it. In return for these benefits, a reader must be prepared to repeat its critical section if it discovers, at the end, that it has overlapped the execution of a writer. Moreover, the reader's actions must be simple enough that nothing a writer might do can cause the reader to experience an unrecoverable error—divide by zero, dereference of an invalid pointer, infinite loop, etc. Put another way, seqlocks provide mutual exclusion among writers, but not between readers and writers.

```
class seqlock
    int n := 0

int seqlock.reader_start():
    int seq
    repeat                                  // spin until even
        seq := n
    until seq ≡ 0 mod 2
    fence(RR)
    return seq

bool seqlock.reader_validate(int seq):
    fence(RR)
    return (n == seq)

bool seqlock.become_writer(int seq):
    fence(RR)
    if CAS(&n, seq, seq+1)
        fence(RW)
        return true
    return false

seqlock.writer_acquire():
    loop
        int seq := n
        if seq ≡ 0 mod 2
            if CAS(&n, seq, seq+1) break
    fence(RR, RW)

seqlock.writer_release():
    fence(RW, WW)
    n++
```

Figure 6.6: Centralized implementation of a sequence lock.

Rather, they allow a reader to discover, after the fact, that its execution may not have been valid, and needs to be retried.

A simple, centralized implementation of sequence locks appears in Figure 6.6. The lock is represented by single integer. An odd value indicates that the lock is held by a writer; an even value indicate that it is not. For writers, the integer behaves like a test-and-test_and_set lock. We assume that writers are rare.

A reader spins until the lock is even, and then proceeds, remembering the value it saw. If it sees the same value in reader_validate, it knows that no writer has been active, and that everything it has read in its critical section is mutually consistent. (We assume that critical sections are short enough—and writers rare enough—that n can never roll over and repeat a value before the reader completes. For real-world integers and critical sections,

this is a completely safe assumption.) If a reader sees a different value in validate, however, it knows that it has overlapped a writer and must repeat its critical section.

```
repeat
    int s := SL.reader_start()
    // critical section
until SL.reader_validate(s)
```

It is essential here that the critical section be *idempotent*—harmlessly repeatable. In the canonical use case, seqlocks serve in the Linux kernel to protect multi-word time information, which can then be read atomically and consistently. If a reader critical section updates thread-local data (only shared data must be read-only), the idiom shown above can be modified to undo the updates in the case where reader_validate returns false.

If a reader needs to perform a potentially "dangerous" operation (integer divide, pointer dereference, unbounded iteration, memory allocation/deallocation, etc.) within its critical section, the reader_validate method can be called repeatedly (with the same parameter each time). If reader_validate returns true, the upcoming operation is known to be safe (all values read so far are mutually consistent); if it returns false, consistency cannot be guaranteed, and code should branch back to the top of the repeat loop. In the (presumably rare) case where a reader discovers that it really needs to write, it can request a "promotion" with become_writer:

```
loop
    int s := SL.reader_start()
    ...
    if unlikely_condition
        if !become_writer(s) continue          // return to top of loop
        ...
        writer_release()
    else                                       // still reader
        ...
        if reader_validate(s) break
```

After becoming a writer, of course, a thread has no further need to validate its reads: it will exit the loop above after calling writer_release.

Unfortunately, because they are inherently speculative, seqlocks induce a host of data races [Boehm, 2012]. Every read of a shared location in a reader critical section will typically race with some write in a writer critical section. In a language like C++, which forbids data races, a straightforward fix is to label all read locations atomic; this will prevent the compiler from reordering accesses, and cause it to issue memory fence instructions that prevent the hardware from reordering them either. This solution is overly conservative, however: it inhibits reorderings that are clearly acceptable within idempotent read-only critical sections. Boehm [2012] explores the data-race issue in depth, and describes other, less conservative options.

Together, the problems of inconsistency and data races are subtle enough that seqlocks are best thought of as a special-purpose technique, to be employed by experts in well constrained circumstances, rather than as a general-purpose form of synchronization. That said, seqlock usage can be safely automated by a compiler that understands the nature of speculation. Dalessandro et al. [2010a] describe a system (in essence, a minimal implementation of transactional memory) in which (1) a global sequence lock serializes all writer transactions, (2) fences and reader_validate calls are inserted automatically where needed, and (3) local state is checkpointed at the beginning of each reader transaction, for restoration on abort. A follow-up paper [Dalessandro et al., 2010c] describes a more concurrent system, in which writer transactions proceed speculatively, and a global sequence lock serializes only the write-back of buffered updates. We will return to the subject of transactional memory in Chapter 9.

6.3 READ-COPY UPDATE

Read-copy update, more commonly known as simply RCU [McKenney, 2004; McKenney et al., 2001], is a synchronization strategy that attempts to drive the overhead of reader synchronization as close to zero as possible, at the expense of potentially very high overhead for writers. Instances of the strategy typically display the following four main properties:

no shared updates by readers. As in a sequence lock, readers modify no shared metadata before or after performing an operation. While this makes them invisible to writers, it avoids the characteristic cache misses associated with locks. To ensure a consistent view of memory, readers may need to execute RR fences on some machines, but these are typically much cheaper than a cache miss.

single-pointer updates. Writers use a lock to synchronize with one another. They make their updates visible to readers by performing a single atomic memory update—typically by "swinging" a pointer to refer to the new version of (some part of) a data structure, rather than to the old version. Readers serialize before or after the writer depending on whether they see this update.

unidirectional data traversal. To ensure serializability, readers must never inspect a pointer more than once. Moreover if writers A and B modify different pointers, and A serializes before B, it must be impossible for any reader to see B's update but not A's. The most straightforward way to ensure this is to require all structures to be trees, traversed from the root toward the leaves, and by arranging for writers to replace entire subtrees.

delayed reclamation of deallocated data. When a writer updates a pointer, readers that have already dereferenced the old version—but have not yet finished their operations—may continue to read old data for some time. Implementations of RCU

must therefore provide a (potentially conservative) way for writers to tell that all readers that could still access old data have finished their operations and returned. Only then can the old data's space be reclaimed.

Implementations and applications of RCU vary in many details, and may diverge from the description above if the programmer is able to prove that (application-specific) semantics will not be compromised. We consider relaxations of the single-pointer update and unidirectional traversal properties below. First, though, we consider ways to implement relaxed reclamation and to accommodate, at minimal cost, machines with relaxed memory order.

Grace Periods and Relaxed Reclamation. In a language and system with automatic garbage collection, the delayed reclamation property is trivial: the normal collector will reclaim old data versions when—and only when—no readers can see them any more. In the more common case of manual memory management, a writer may wait until all readers of old data have completed, and then reclaim space itself. Alternatively, it may append old data to a list for eventual reclamation by some other, bookkeeping thread. The latter option reduces latency for writers, potentially improving performance.

Arguably the biggest differences among RCU implementations concern the "grace period" mechanism used (in the absence of a general-purpose garbage collector) to determine when all old readers have completed. In a nonpreemptive OS kernel (where RCU was first employed), the writer can simply wait until a (voluntary) context switch has occurred on every core. Perhaps the simplest way to do this is to request migration to each core in turn: such a request will be honored only after any active reader on the target core has completed.

More elaborate grace period implementations can be used in more general contexts. Desnoyers et al. [2012, App. D] describe several implementations suitable for user-level applications. Most revolve around a global counter C and a global set S of counters, indexed by thread id. C is monotonically increasing (extensions can accommodate rollover): in the simplest implementation, it is incremented at the end of each write operation. In a partial violation of the no-shared-updates property, S is maintained by readers. Specifically, $\mathsf{S}[i]$ will be zero if thread i is not currently executing a reader operation. Otherwise, $\mathsf{S}[i]$ will be j if C was J when thread i's current reader operation began. To ensure a grace period has passed (and all old readers have finished), a writer iterates through S, waiting for each element to be either zero or a value greater than or equal to the value just written to C. Assuming that each set element lies in a separate cache line, the updates performed by reader operations will usually be cache hits, with almost no performance impact.

Memory Ordering. When beginning a user-level read operation with grace periods based on the global counter and set, a thread must issue a WR fence after updating its entry in S. At the end of the operation, it must issue a RW fence before updating its entry.

Within the read operation, it must issue a RR fence after dereferencing a pointer that might be updated by a writer. Among these fences, the WR case is typically the most expensive (the others will in fact be no-ops on a TSO machine). We can eliminate the WR fence in the common case by requiring the writer to interrupt all potential readers (e.g., with a Posix signal) at the end of a write operation. The signal handler can then "handshake" with the writer, with appropriate memory barriers, thereby ensuring (a) that each reader's update of its element in S is visible to the writer, and (b) that the writer's updates to shared data are visible to all readers. Assuming that writers are rare, the cost of the signal handling will be outweighed by the no-longer-required WR fences in the (much more numerous) reader operations.

Multi-Write Operations The single-pointer update property may become a single-*word* update in data structures that use some other technique for traversal and space management. In many cases, it may also be possible to accommodate multi-word updates by exploiting application-specific knowledge. In their original paper, for example, McKenney et al. [2001] presented an RCU version of doubly-linked lists, in which a writer must update both forward and backward pointers. The secret here is that readers only search for nodes in the forward direction; the backward pointers simply facilitate constant-time deletion. Readers therefore serialize before or after a writer depending on whether they see the change to the forward pointer.

In general, readers may need to see a *set* of updates atomically. One can always ensure this by copying the entire data structure, updating the copy, and then swinging a single root pointer, but the copy will be very inefficient when the structure is large and the number of changes is small. Interestingly, as noted by Clements et al. [2012], a similar problem occurs in functional programming languages, where one often needs to create a new (logically immutable) version of a large object that differs from the old in only a few locations. Drawing on techniques developed in the functional programming community, Clements et al. show how to implement an RCU version of the balanced binary trees used to represent address spaces in the Linux kernel. They begin by noting that an insertion or deletion in the tree can be made by changing a single pointer in a leaf. Any needed rebalancing can be delegated to separate operations. Moreover the rebalancing is semantically neutral: while costs may differ slightly, a reader is guaranteed to obtain the same result regardless of whether it runs before or after the rebalance operation (so long as it is atomic).

In trees that are rebalanced with *rotation* operations, one can show that any given rotation will change only a small internal subtree with a single incoming pointer. One can thus effect an RCU rotation by creating a new version of this subtree and then swinging the pointer to its root. To rotate nodes x and z in Figure 6.7, for example, one creates new nodes x′ and z′, initializes their child pointers to refer to trees A, B, and C, and uses a CAS to swing p's left or right child pointer (as appropriate) to refer to x′ instead of z. Once a

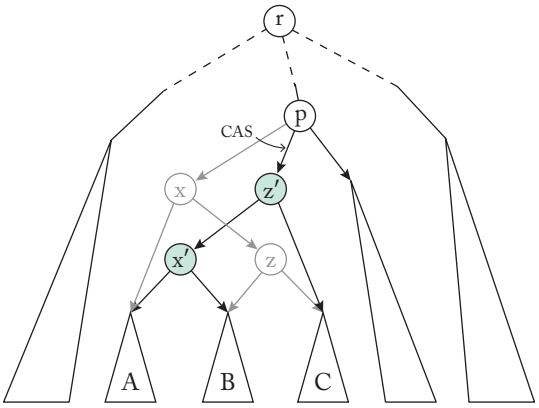

Figure 6.7: Rebalancing of a binary tree via internal subtree replacement (rotation). Adapted from Clements et al. [2012, Fig. 8(b)]. Prior to the replacement, node z is the right child of node x. After the replacement, x′ is the left child of z′.

grace period has expired, nodes x and z can be reclaimed. In the meantime, readers that have traveled through x and z will still be able to search correctly down to the fringe of the tree.

In-Place Updates As described above, RCU is designed to incur essentially no overhead for readers, at the expense of very high overhead for writers. In some cases, however, even this property can be relaxed. In the same paper that introduced RCU balanced trees, Clements et al. [2012] observe that trivial updates to page tables—specifically, single-leaf modifications associated with demand page-in—are sufficiently common to be a serious obstacle to scalability on large shared-memory multiprocessors. Their solution is essentially a hybrid of RCU and sequence locks. Major (multi-page) update operations continue to function as RCU writers: they exclude one another in time, install their changes via single-pointer update, and wait for a grace period before reclaiming no-longer-needed space. The page fault interrupt handler, however, functions as an RCU reader. If it needs to modify a page table entry to effect demand page in, it makes its modifications in place.

This convention introduces a variety of synchronization challenges. For example: a fault handler that overlaps in time with a major update (e.g., an munmap operation that invalidates a broad address range) may end up modifying the about-to-be-reclaimed version of a page table entry, in which case it should not return to the user program as if nothing had gone wrong. If each major update acquires and updates a (per-address-space) sequence lock, however, then the fault handler can check the value of the lock both before and after its operation. If the value has changed, it can retry—perhaps acquiring the lock itself if starvation might be a concern. Similarly, if fault handlers cannot safely run concurrently

with one another (e.g., if they need to modify more than one word in memory), then they need their own synchronization—perhaps a separate sequence lock in each page table entry.

Bibliography

Nagi M. Aboulenein, James R. Goodman, Stein Gjessing, and Philip J. Woest. Hardware support for synchronization in the scalable coherent interface (SCI). In *Proceedings of the Eighth International Parallel Processing Symposium (IPPS)*, pages 141–150, Cancun, Mexico, April 1994.

Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Justin Gottschlich, et al. *Draft Specification of Transactional Language Constructs for C++, Version 1.1.* Intel, Oracle, IBM, and Red Hat, February 2012.

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the Seventeenth International Symposium on Computer Architecture (ISCA)*, pages 2–14, Seattle, WA, May 1990.

Sarita V. Adve, Vijay S. Pai, and Parthasarathy Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. *Proceedings of the IEEE*, 87 (3):445–455, 1999.

Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann, San Francisco, CA, 2002.

George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin Cummings, Redwood City, CA, 1989.

Noga Alon, Amnon Barak, and Udi Manber. On disseminating information reliably without broadcasting. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 74–81, Berlin, Germany, September 1987.

Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Experience with k42, an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the Thirty-eighth ACM Symposium on Principles of Programming Languages (POPL)*, pages 487–498, Austin, TX, January 2011.

Marc A. Auslander, David Joel Edelsohn, Orran Yaakov Krieger, Bryan Savoye Rosenburg, and Robert W. Wisniewski. Enhancement to the MCS lock for increased functionality and improved programmability. U. S. patent application number 20030200457 (abandoned), October 2003.

David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The 'double-checked locking is broken' declaration, 2001. `www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

Rudolf Bayer and Mario Schkolnick. Concurrency of operations on *B*-trees. *Acta Informatica*, 9(1):1–21, 1977.

Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2006.

Hans-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 12–20, Beijing, China, June 2012.

Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, Tucson, AZ, June 2008.

Eugene D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, August 1986.

Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Cătălin Roşu, and Ray Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, Santa Barbara, CA, July 1995.

James E. Burns and Nancy A. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the Eighteenth Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, Monticello, IL, October 1980. A revised version of this paper was published as "Bounds on Shared memory for Mutual Exclusion", *Information and Computation*, 107(2):171–184, December 1993.

Dan Chazan and Willard L. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, April 1969.

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, London, United Kingdom, March 2012.

Cliff Click Jr. And now some hardware transactional memory comments. Author's Blog, Azul Systems, February 2009. `http://www.azulsystems.com/blog/cliff/2009-02-25-and-now-some-hardware-%transactional-memory-comments`.

Edward G. Coffman, Jr., Michael J. Elphick, and Arie Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

Pierre-Jacques Courtois, F. Heymans, and David L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, October 1971.

Travis S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington Computer Science Department, February 1993.

David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1998. With Anoop Gupta.

Luke Dalessandro, Dave Dice, Michael L. Scott, Nir Shavit, and Michael F. Spear. Transactional mutex locks. In *Proceedings of the Sixteenth International Euro-Par Conference*, pages II:2–13, Ischia-Naples, Italy, August–September 2010a.

Luke Dalessandro, Michael L. Scott, and Michael F. Spear. Transactions as the foundation of a memory consistency model. In *Proceedings of the Twenth-Fourth International Symposium on Distributed Computing (DISC)*, pages 20–34, Cambridge, MA, September 2010b.

Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the Fifteenth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, Bangalore, India, January 2010c.

Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, February 2012.

Dave Dice, Hui Huang, and Mingyao Yang. Asymmetric dekker synchronization. Lead author's blog, July 2001. `blogs.oracle.com/dave/resource/Asymmetric-Dekker-Synchronization.txt`.

Dave Dice, Mark Moir, and William N. Scherer III. Quickly reacquirable locks. Technical Report Technical Report, Sun Microsystems Laboratories, 2003. Subject of U.S. Patent #7,814,488.

Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the Fourteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 157–168, Washington, DC, March 2009.

David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. In *Proceedings of the Seventeenth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 247–256, New Orleans, LA, February 2012.

Edsger W. Dijkstra. Een algorithme ter voorkoming van de dodelijke omarming. Technical Report EWD-108, IBM T. J. Watson Research Center, early 1960s. In Dutch. Circulated privately.

Edsger W. Dijkstra. The mathematics behind the banker's algorithm. In *Selected Writings on Computing: A Personal Perspective*, pages 308–312. Springer-Verlag, 1982.

Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

Edsger W. Dijkstra. The structure of the 'THE' multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968a.

Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, United Kingdom, 1968b. Originally EWD-123, Technological University of Eindhoven, 1965.

Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19 (11):624–633, November 1976.

Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure. In *Proceedings of the Twentieth International Symposium on Computer Foundations of Computer Science (FOCS)*, pages 234–254, San Juan, Puerto Rico, October 1979.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

Nissim Francez. *Fairness*. Springer-Verlag, 1986.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, PQ, Canada, June 1998.

James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the Tenth International Symposium on Computer Architecture (ISCA)*, pages 124–131, Stockholm, Sweden, June 1983.

James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, Boston, MA, April 1989.

Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer: Designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, 32(2):175–189, February 1983.

Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.

Rajiv Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the Third International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 54–63, Boston, MA, April 1989.

Rajiv Gupta and Charles R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3): 161–180, June 1989.

Yijie Han and Raphael A. Finkel. An optimal scheme for disseminating information. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages II:198–203, University Park, PA, August 1988.

Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, San Francisco, CA, second edition, 2010. First edition, by Larus and Rajwar only, 2007.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the Sixteenth International Symposium on Distributed Computing (DISC)*, pages 265–279, Toulouse, France, October 2002.

Debra A. Hensgen, Raphael A. Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, February 1988.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth International Symposium on Computer Architecture (ISCA)*, pages 289–300, San Diego, CA, May 1993.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, San Francisco, CA, 2008.

Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the Sixteenth International Symposium on Distributed Computing (DISC)*, pages 339–353, Toulouse, France, October 2002.

Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, Providence, RI, May 2003a.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, Boston, MA, July 2003b.

Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3): 463–492, July 1990.

Mark D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, August 1998.

IBM, 1981. *System/370 Principles of Operation*. IBM Corporation, ninth edition, October 1981.

Jean Ichbiah, John G. P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, Cambridge, England, 1991. ISBN 0-521-39267-5.

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, December 2011. Order number 325462-041US.

Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the Twenty-second ACM Symposium on Principles of Distributed Computing (PODC)*, pages 285–294, Boston, MA, July 2003.

Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

JRuby. Community website. `jruby.org/`.

Jython. Project website. `jython.org/`.

KSR. *KSR1 Principles of Operation*. Kendall Square Research, Waltham, MA, 1992.

Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the Seventeenth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 141–150, New Orleans, LA, February 2012.

Eric Koskinen and Maurice Herlihy. Dreadlocks: Efficient deadlock detection. In *Proceedings of the Twentieth International Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 297–303, Munich, Germany, June 2008.

Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, October 1988.

Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Proceedings of the Gelato Federation Meeting*, San Jose, CA, May 2005.

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

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

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

Doug Lea. The JSR-133 cookbook for compiler writers, March 2001. `g.oswego.edu/dl/jmm/cookbook.h`.

Craig A. Lee. Barrier synchronization over multistage interconnection networks. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 130–133, Dallas, TX, December 1990.

Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

Boris D. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages II:175–179, University Park, PA, August 1989.

Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

Peter Magnussen, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium (IPPS)*, pages 165–171, Cancun, Mexico, April 1994.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the Thirty-second ACM Symposium on Principles of Programming Languages (POPL)*, pages 378–391, Long Beach, CA, January 2005.

Virendra J. Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the Thirteenth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 227–236, Salt Lake City, UT, February 2008.

Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the Nineteenth International Symposium on Distributed Computing (DISC)*, pages 354–368, Cracow, Poland, September 2005.

Evangelos P. Markatos. Multiprocessor synchronization primitives with priorities. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 1–7, Atlanta, GA, May 1991.

Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering, Oregon Health and Science University, July 2004.

Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russel, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Proceedings of the Ottawa Linux Symposium*, pages 338–367, Ottawa, ON, Canada, July 2001.

John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 106–113, Williamsburg, VA, April 1991a.

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

Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proceedings of the Fourteenth International Symposium on Distributed Computing (DISC)*, pages 164–178, Toledo, Spain, October 2000.

Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):491–504, August 2004a.

Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC23089, IBM T. J. Watson Research Center, January 2004b.

Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

Mark Moir and Nir Shavit. Concurrent data structures. In Dinesh P. Metha and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, page Chapter 47. Chapman and Hall / CRC Press, San Jose, CA, 2005.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the Eleventh International Symposium on Computer Architecture (ISCA)*, pages 348–354, 1984.

Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the Ninth ACM Symposium on the Theory of Computing (STOC)*, pages 91–97, Boulder, CO, May 1977.

Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA)*, pages 241–252, Anaheim, CA, February 2003.

Zoran Radović and Erik Hagersten. Efficient synchronization for nonuniform communication architectures. In *Proceedings, Supercomputing 2002 (SC)*, pages 1–13, Baltimore, MD, November 2002.

David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the Eleventh International Symposium on Computer Architecture (ISCA)*, pages 340–347, Ann Arbor, MI, June 1984.

Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the Twenty-first Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 263–272, Portland, OR, October 2006.

Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, 1997.

Michael L. Scott and John M. Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *International Journal of Parallel Programming*, 22(4):449–481, August 1994.

Michael L. Scott and Maged M. Michael. The topological barrier: A synchronization abstraction for regularly-structured parallel applications. Technical Report TR 605, Department of Computer Science, University or Rochester, January 1996.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

Jun Shirako, David Peixotto, Vivek Sarkar, and William N. Scherer III. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 277–288, Island of Kos, Greece, June 2008.

Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool, San Francisco, CA, 2011.

Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.

Håkan Sundell and Philippas Tsigas. NOBLE: Non-blocking programming support via lock-free shared abstract data types. *Computer Architecture News*, 36(5):80–87, December 2008.

Peiyi Tang and Pen-Chung Yew. Software combining algorithms for distributing hot-spot addressing. *Journal of Parallel and Distributed Computing*, 10(2):130–139, 1990.

Gadi Taubenfeld. Shared memory synchronization. *Bulletin of the European Association for Theoretical Computer Science (BEATCS)*, (96):81–103, October 2008.

Gadi Taubenfeld. The black-white bakery algorithm. In *Proceedings of the Eighteenth International Symposium on Distributed Computing (DISC)*, pages 56–70, Amsterdam, The Netherlands, October 2004.

Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson Education–Prentice-Hall, 2006.

R. Kent Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 65–74, Vienna, Austria, September 2010.

William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, February 1989.

Philip J. Woest and James R. Goodman. An analysis of synchronization mechanisms in shared memory multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing (ISSMM)*, pages 152–165, Toyko, Japan, April 1991.

Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2): 28–40, April 1996.

Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 36(4):388–395, April 1987.

Author's Biography

MICHAEL L. SCOTT

Michael L. Scott is a Professor and past Chair of the Department of Computer Science at the University of Rochester. He received his Ph.D. from the University of Wisconsin–Madison in 1985. His research interests span operating systems, languages, architecture, and tools, with a particular emphasis on parallel and distributed systems. He is best known for work in synchronization algorithms and concurrent data structures, in recognition of which he shared the 2006 SIGACT/SIGOPS Edsger W. Dijkstra Prize. Other widely cited work has addressed parallel operating systems and file systems, software distributed shared memory, and energy-conscious operating systems and microarchitecture. His textbook on programming language design and implementation (*Programming Language Pragmatics*, third edition, Morgan Kaufmann, Feb. 2009) is a standard in the field. In 2003 he served as General Chair for *SOSP*; more recently he has been Program Chair for *TRANSACT '07*, *PPoPP '08*, and *ASPLOS '12*. He was named a Fellow of the ACM in 2006 and of the IEEE in 2010. In 2001 he received the University of Rochester's Robert and Pamela Goergen Award for Distinguished Achievement and Artistry in Undergraduate Teaching.