Synthesis Lectures on
Computer Architecture

SYNTHESIS
COLLECTION OF TECHNOLOGY

Michael L. Scott
Trevor Brown

Shared-Memory Synchronization

Second Edition

Springer

Synthesis Lectures on Computer Architecture

Series Editor

Natalie Enright Jerger, University of Toronto, Toronto, Canada

This series covers topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

Michael L. Scott · Trevor Brown

Shared-Memory Synchronization

Second Edition

Springer

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

Trevor Brown
University of Waterloo
Waterloo, ON, Canada

ISSN 1935-3235 ISSN 1935-3243 (electronic)
Synthesis Lectures on Computer Architecture
ISBN 978-3-031-38683-1 ISBN 978-3-031-38684-8 (eBook)
https://doi.org/10.1007/978-3-031-38684-8

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Kelly and Britt

Preface

This monograph grows out of nearly 40 years of experience in synchronization and concurrent data structures. Though written primarily from the perspective of systems software, it reflects our conviction that the field cannot be understood without a solid grounding in both concurrency theory and computer architecture.

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 synchronizing code, and Chapter 3 introduces formal concepts that illuminate issues of feasibility and correctness.

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 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.)

Given the volume of material, readers with limited time may wish to sample topics of particular interest. All readers, however, should make sure they are familiar with the material in Chapters 1 through 3. In our experience, practitioners often underestimate the value of formal foundations, and theoreticians are sometimes vague about the nature and impact of architectural constraints. Readers may also wish to bookmark Table 2.1 (page 21), which describes the memory model assumed by the pseudocode. Beyond that:

- Computer architects interested in the systems implications of modern multicore hardware may wish to focus on Secs. 2.2–2.3.1, 3.3–3.4, 4.2–4.3, 4.5.1, 5.1–5.2, 8.1–8.18.2.1, and 9.2.
- Programmers with an interest in operating systems and run-time packages may wish to focus on Secs. 2.2–2.3.1, all of Chapters 3–6, and Sec. 7.5.

- Authors of parallel libraries may wish to focus on Secs. 2.2–2.3.1 and 5.4, plus all of Chapters 3, 7, and 8.
- Compiler writers will need to understand all of Chapters 2 and 3, plus Secs. 4.5.2, 5.1, 5.3.1, 5.3.3, and 7.3–7.4.

Some readers may be surprised to find that the monograph contains no concrete performance results. This omission reflects a deliberate decision to focus on qualitative comparisons among algorithmic alternatives. Performance is obviously of great importance in the evaluation of synchronization mechanisms and concurrent data structures (and our papers are full of hard numbers), but the constants change with time, and they depend in many cases on characteristics of the specific application, language, operating system, and hardware at hand. When relative performance is in doubt, system designers would be well advised to benchmark the alternatives in their own particular environment.

This second edition, published roughly a decade after the original, has numerous small improvements throughout—clarifications, bug fixes, and references to newer work. It also incorporates two major updates.

First, we have re-worked and clarified the memory model and notation (Secs. 2.2 and 2.3) used in our examples. In particular, we now employ explicit load and store operations for all accesses to shared (atomic) variables, and we have adopted the C++ convention of making such accesses fully ordered (sequentially consistent) unless otherwise noted. These conventions necessitated changes to most of our algorithmic pseudocode.

Second, we have added significant new material, particularly in Chapter 8 (Nonblocking Algorithms). This includes detailed description of the EFRB tree (Sec. 8.6.1); a survey of other trees (Sec. 8.6.2); additional lists, queues, deques, hash tables, and skip lists (Secs. 8.2–8.5); extensive discussion of safe memory reclamation (Sec. 8.7); and expanded coverage of high-level atomic constructions (Sec. 8.10).

The field, of course, continues to evolve, but the fundamentals remain. We hope our work helps others to appreciate their beauty and lasting utility.

Rochester, NY, USA — Michael L. Scott
Waterloo, ON, Canada — Trevor Brown
Summer 2023

The original version of this book has been revised: Author provided text corrections and figure corrections has been incorporated in the following chapters: Chapter 3 Fig. 3.1, Chapter 4 Fig 4.14 and Chapter 8 Figs. 8.2 and 8.8. A correction to this book can be found at https://doi.org/10.1007/978-3-031-38684-8_10

Acknowledgments

This monograph has benefited from the feedback of many generous colleagues. Sarita Adve, Hans Boehm, Dave Dice, Maurice Herlihy, Mark Hill, Victor Luchangco, Paul McKenney, Maged Michael, Nir Shavit, and Mike Swift all read through draft material, and made numerous helpful suggestions for improvements. We are particularly indebted to Hans for his coaching on memory consistency models, to Victor for his careful vetting of Chapter 3, and to Peter Buhr for detailed guidance on ordering constraints in various algorithms. (The mistakes that remain are of course our own!) Our thanks as well to the students of Mark's CS758 course in the fall of 2012, who provided additional feedback, and to Dong Chen, Jakub Łopuszański, and Jinjian Ma, who found additional errors in the first edition. Finally, our thanks to Mark and to Mike Morgan for convincing Michael to undertake the first edition, and to Margaret Martonosi and Natalie Enright Jerger for their skillful ongoing stewardship of the Synthesis Series on Computer Architecture.

Rochester, NY, USA — Michael L. Scott
Waterloo, ON, Canada — Trevor Brown
Summer 2023

Contents

About the Authors

Michael L. Scott is the Arthur Gould Yates Professor of Engineering and 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. His textbook on programming language design and implementation (*Programming Language Pragmatics*, fourth edition, Morgan Kaufmann, 2016; fifth edition forthcoming) is a standard in the field. He served as General Chair of *SOSP* in 2003 and as Program Chair of *TRANSACT*'07, *PPoPP'08*, and *ASPLOS'12*. He was named a Fellow of the ACM in 2006, of the IEEE in 2010, and of the AAAS in 2021. At the University of Rochester, he received the Robert and Pamela Goergen Award for Distinguished Achievement and Artistry in Undergraduate Teaching in 2001, the Edmund A. Hajim School of Engineering Lifetime Achievement Award in 2018, and the William H. Riker University Award for Graduate Teaching in 2020.

Trevor Brown is an Assistant Professor in the Cheriton School of Computer Science at the University of Waterloo. He completed his Ph.D. under the supervision of Faith Ellen at the University of Toronto in 2017, and conducted postdoctoral studies at the Institute of Science and Technology, Austria, and the Technion, Israel. His research straddles theory and practice, and focuses on the question of how large-scale multicore systems can be programmed easily, efficiently, and correctly. His specific research interests include concurrent data structures, lock-free synchronization, memory management, transactional memory, and non-volatile memory. At the University of Toronto, he won the Award for Excellence in Teaching Assistance in 2014. He has served multiple times on the Program Committees of *ICDCS, PODC, PPoPP, SIROCCO*, and *SPAA*, and was Publication Chair for *PPoPP'19*. He received the Best Paper Award at *PPoPP'20*, Best Artifact Awards at *PPoPP'21* and *PPoPP'22*, and Finalist status in the Best Paper competitions at *PPoPP'21* and *SPAA'22*.

Introduction

1

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 instruction steps: 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 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.

M. L. Scott and T. Brown, *Shared-Memory Synchronization*, Synthesis Lectures on Computer Architecture, https://doi.org/10.1007/978-3-031-38684-8_1

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

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.

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.

1.1 Atomicity

The example on page 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 indivisible 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.

Concurrency and Parallelism

Sadly, the adjectives "concurrent" and "parallel" are used in different ways by different authors. For some authors (including the current ones), 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*).

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

If thread 1 calls ctr_inc(i) and thread 2 calls ctr_inc(j), we shall 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, 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(int n, int i, int 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 shall need to consider whether the benefit of concurrency in independent operations outweighs the cost of the extra lock. More significantly, we shall 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 ways to formalize this requirement, most notably *linearizability* and several variants of *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 shall need each insert or remove operation to execute atomically. More than this, however, we shall 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.insert(data d):
    atomic
        await ¬Q.full()
        // put d in next empty slot
```

```
data Q.remove():
    atomic
        await ¬Q.empty()
        // return data from next full 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 Sec. 3.3, atomicity is more difficult to implement, in a formal, theoretical sense.

1.3 Spinning Versus 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 isolated 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*[1]:

```
type lock = bool := false

L.acquire():                    L.release():
    while L.TAS()                   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

[1] As we shall see in Secs. 2.2 and 3.3, both of the examples in this section—for condition synchronization and for mutual exclusion—would in practice need to be extended with *ordering annotations* that prevent the compiler and hardware from performing optimizations that are unsafe in multithreaded code. Correctly annotated versions of these examples can be found on Secs. 5.1 and 4.1.1, respectively.

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.

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." The Mach operating system uses "task" where we use "process." And some systems introduce additional words—e.g., "activation," "fiber," "filament," or "hart."

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—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, and Q is some other predicate on states. 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; we consider these briefly in Sec. 3.2.2.

Multiple Meanings of "Blocking"

"Blocking" is another word with more than one meaning. In this chapter, we are using it in an implementation-oriented sense, as a synonym for "de-scheduling" (giving the underlying kernel thread or hardware core to another user or kernel thread). In a similar vein, it is sometimes used in a "systems" context to refer to an operation (e.g., a "blocking" I/O request) that waits for a response from some other system component. In Chapter 3, we will use it in a more formal sense, as a synonym for "unable to make forward progress on its own." To a theoretician, a thread that is spinning on a condition that must be made true by some other thread is just as "blocked" as one that has given up its kernel thread or hardware core, and will not run again until some other thread tells the scheduler to resume it. Which definition we have in mind should usually be clear from context.

Most of our discussions of correctness will focus on safety properties. Interestingly, *deadlock freedom*, which one might initially imagine to be a matter of liveness, is actually one of safety: because deadlock can be described as a predicate that takes the current system state as input, deadlock freedom simply insists that the predicate be false in all reachable states.

Architectural Background

2

The correctness and performance of synchronization algorithms depend crucially on 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 synchronizing instructions (ordered accesses, memory fences, 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. Symmetric machines are sometimes said to have a *uniform memory access* (UMA) architecture. More common today are *nonuniform memory access* (NUMA) machines, in which each memory bank is associated with a processor (or in 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 2–16

M. L. Scott and T. Brown, *Shared-Memory Synchronization*, Synthesis Lectures on Computer Architecture, https://doi.org/10.1007/978-3-031-38684-8_2

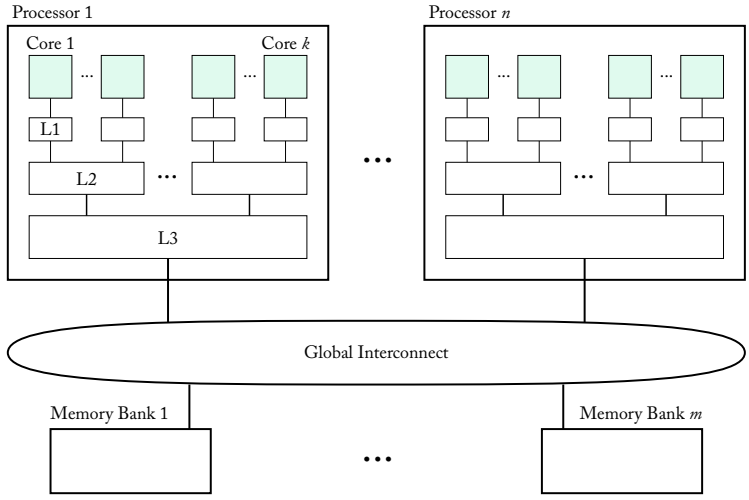

Figure 2.1 Typical symmetric (uniform memory access—UMA) machine. Numbers of components of various kinds, and degree of sharing at various levels, differs across manufacturers and models.

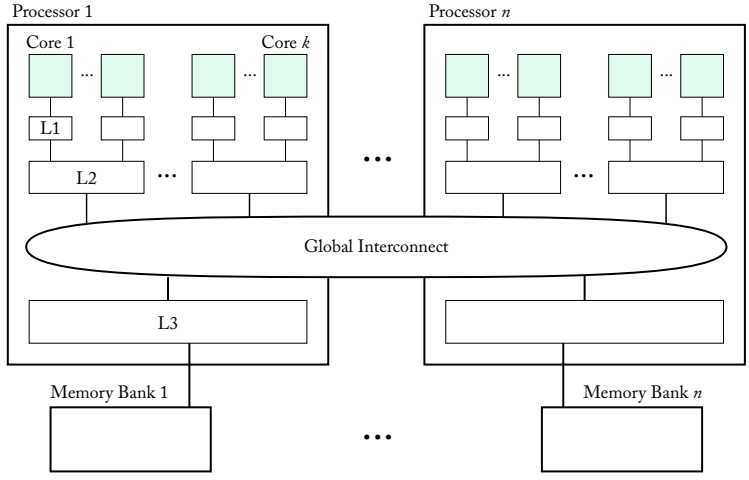

Figure 2.2 Typical nonuniform memory access (NUMA) machine. Again, numbers of components of various kinds, and degree of sharing at various levels, differs across manufacturers and models.