APPENDIX

Hardware basics

B

A novice was trying to fix a broken Lisp machine by turning the power off and on. Knight, seeing what the student was doing spoke sternly: "You cannot fix a machine just by power-cycling it with no understanding of what is going wrong." Knight turned the machine off and on. The machine worked.

(From "AI Koans," a collection of jokes popular at MIT in the 1980s.)

B.1 Introduction (and a puzzle)

You can do a pretty good job of programming a uniprocessor without understanding much about computer architecture, but the same is not true of multiprocessors. You cannot program a multiprocessor effectively unless you know what a multiprocessor *is*. We illustrate this point by a puzzle. We consider two programs that are logically equivalent, but one is much less efficient than the other. Ominously, the simpler program is the inefficient one. This discrepancy cannot be explained, nor the danger avoided, without a basic understanding of modern multiprocessor architectures.

Here is the background to the puzzle. Suppose two threads share a resource that can be used by only one thread at a time. To prevent concurrent use, each thread must *lock* the resource before using it, and *unlock* it afterward. We study many ways to implement locks in Chapter 7. For the puzzle, we consider two simple implementations in which the lock is a single Boolean field. If the field is *false*, the lock is free, and otherwise it is in use. We manipulate the lock with the `getAndSet(`v`)` method, which atomically swaps its argument v with the field value. To acquire the lock, a thread calls `getAndSet(`*true*`)`. If the call returns *false*, then the lock was free, and the caller succeeded in locking the object. Otherwise, the object was already locked, and the thread must try again later. A thread releases a lock simply by storing *false* into the Boolean field.

In Fig. B.1, the *test-and-set* (`TASLock`) lock repeatedly calls `getAndSet(`*true*`)` (line 4) until it returns *false*. By contrast, in Fig. B.2, the *test-and-test-and-set* lock (`TTASLock`) repeatedly reads the lock field (by calling `state.get()` at line 5) until it returns *false*, and only then calls `getAndSet()` (line 6). It is important to understand that reading the lock value is atomic, and applying `getAndSet()` to the lock value is atomic, but the combination is not atomic: Between the time a thread reads the lock value and the time it calls `getAndSet()`, the lock value may have changed.

```
1  public class TASLock implements Lock {
2    ...
3    public void lock() {
4      while (state.getAndSet(true)) {} // spin
5    }
6    ...
7  }
```

FIGURE B.1

The TASLock class.

```
1  public class TTASLock implements Lock {
2    ...
3    public void lock() {
4      while (true) {
5        while (state.get()) {}; // spin
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   ...
11 }
```

FIGURE B.2

The TTASLock class.

Before you proceed, you should convince yourself that the TASLock and TTASLock algorithms are logically the same. The reason is simple: In the TTASLock algorithm, reading that the lock is free does not guarantee that the next call to getAndSet() will succeed, because some other thread may have acquired the lock in the interval between reading the lock and trying to acquire it. So why bother reading the lock before trying to acquire it?

Here is the puzzle: While the two lock implementations may be logically equivalent, they perform very differently. In a classic 1989 experiment, Anderson measured the time needed to execute a simple test program on several contemporary multiprocessors. He measured the elapsed time for n threads to execute a short critical section one million times. Fig. B.3 shows how long each lock takes, plotted as a function of the number of threads. In a perfect world, both the TASLock and TTASLock curves would be as flat as the ideal curve on the bottom, since each run does the same number of increments. Instead, we see that both curves slope up, indicating that lock-induced delay increases with the number of threads. Curiously, however, the TASLock is much slower than the TTASLock lock, especially as the number of threads increases. Why?

This appendix covers much of what you need to know about multiprocessor architecture to write efficient concurrent algorithms and data structures. Along the way, we will explain the divergent curves in Fig. B.3.

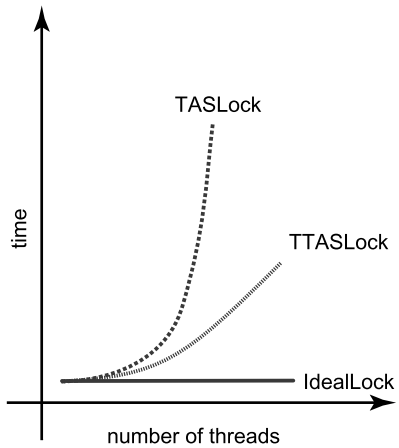

FIGURE B.3

Schematic performance of a `TASLock`, a `TTASLock`, and an ideal lock.

We are concerned with the following components:

- The *processors* are hardware devices that execute software *threads*. There are typically more threads than processors, and each processor runs a thread for a while, sets it aside, and turns its attention to another thread.
- The *interconnect* is a communication medium that links processors to processors and processors to memory.
- The *memory* is actually a hierarchy of components that store data, ranging from one or more levels of small, fast *caches* to a large and relatively slow *main memory*. Understanding how these levels interact is essential to understanding the actual performance of many concurrent algorithms.

From our point of view, one architectural principle drives everything else: *Processors and main memory are far apart*. It takes a long time for a processor to read a value from memory. It also takes a long time for a processor to write a value to memory, and longer still for the processor to be sure that value has actually been installed in memory. Accessing memory is more like mailing a letter than making a phone call. Almost everything we examine in this appendix is the result of trying to alleviate the long time it takes ("high latency") to access memory.

Processor and memory speed change over time, but their *relative* performance changes slowly. Consider the following analogy. Imagine that it is 1980, and you are in charge of a messenger service in mid-town Manhattan. While cars outperform bicycles on the open road, bicycles outperform cars in heavy traffic, so you choose to use bicycles. Even though the technology behind both bicycles and cars has advanced, the *architectural* comparison remains the same. Then, as now, if you are designing an urban messenger service, you should use bicycles, not cars.

B.2 Processors and threads

A multiprocessor consists of multiple hardware *processors*, each of which executes a sequential program. When discussing multiprocessor architectures, the basic unit of time is the *cycle*: the time it takes a processor to fetch and execute a single instruction. In absolute terms, cycle times change as technology advances (from about 10 million cycles per second in 1980 to about 3000 million in 2005), and they vary from one platform to another (processors that control toasters have longer cycles than processors that control web servers). Nevertheless, the relative cost of instructions such as memory access changes slowly when expressed in terms of cycles.

A *thread* is a sequential program. While a processor is a hardware device, a thread is a software construct. A processor can run a thread for a while and then set it aside and run another thread, an event known as a *context switch*. A processor may set aside a thread, or *deschedule* it, for a variety of reasons. Perhaps the thread has issued a memory request that will take some time to satisfy, or perhaps that thread has simply run long enough, and it is time for another thread to make progress. When a thread is descheduled, it may resume execution on another processor.

B.3 Interconnect

The *interconnect* is the medium by which processors communicate with the memory and with other processors. There are essentially two kinds of interconnect architectures in use: *symmetric multiprocessing* (SMP) and *nonuniform memory access* (NUMA). See Fig. B.4.

In an SMP architecture, processors and memory are linked by a *bus* interconnect, a broadcast medium that acts like a tiny ethernet. Both processors and the main memory have *bus controller* units in charge of sending and listening for messages broadcast on the bus. (Listening is sometimes called *snooping*.) SMP architectures are easier to build, but they are not scalable to large numbers of processors because eventually the bus becomes overloaded.

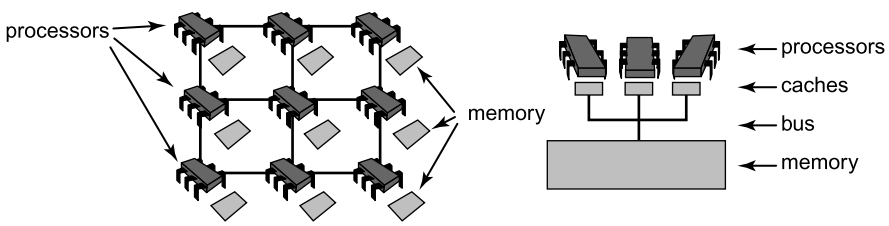

FIGURE B.4

An SMP architecture with caches on the right and a cacheless NUMA architecture on the left.

In a NUMA architecture, a collection of *nodes* are linked by a point-to-point network, like a tiny local area network. Each node contains one or more processors and a local memory. One node's local memory is accessible to the other nodes, and together, the nodes' memories form a global memory shared by all processors. The NUMA name reflects the fact that a processor can access memory residing on its own node faster than it can access memory residing on other nodes. Networks are more complex than buses, and require more elaborate protocols, but they scale better than buses to large numbers of processors.

The division between SMP and NUMA architectures is a simplification: For example, some systems have hybrid architectures, where processors within a cluster communicate over a bus, but processors in different clusters communicate over a network.

From the programmer's point of view, it may not seem important whether the underlying platform is based on a bus, a network, or a hybrid interconnect. It is important, however, to realize that the interconnect is a finite resource shared among the processors. If one processor uses too much of the interconnect's bandwidth, then the others may be delayed.

B.4 Memory

Processors share a *main memory*, which is a large array of *words*, indexed by *address*. The size of a word or an address is platform-dependent, but typically it is either 32 or 64 bits. Simplifying somewhat, a processor reads a value from memory by sending a message containing the desired address to memory. The response message contains the associated *data*, that is, the contents of memory at that address. A processor writes a value by sending the address and the new data to memory, and the memory sends back an acknowledgment when the new data have been installed.

B.5 Caches

On modern architectures, a main memory access may take hundreds of cycles, so there is a real danger that a processor may spend much of its time just waiting for the memory to respond to requests. Modern systems alleviate this problem by introducing one or more *caches*: small memories that are situated closer to the processors and are therefore much faster than main memory. These caches are logically situated "between" the processor and the memory: When a processor attempts to read a value from a given memory address, it first looks to see if the value is already in the cache, and if so, it does not need to perform the slower access to memory. If the desired address's value was found, we say the processor *hits* in the cache, and otherwise it *misses*. In a similar way, if a processor attempts to write an address that is in the cache, it does not need to perform the slower access to memory. The proportion of requests satisfied in the cache is called the cache *hit ratio* (or *hit rate*).

Caches are effective because most programs display a high degree of *locality*: If a processor reads or writes a memory address (also called a memory location), then it is likely to read or write the same location again soon. Moreover, if a processor reads or writes a memory location, then it is also likely to read or write *nearby* locations soon. To exploit this second observation, caches typically operate at a *granularity* larger than a single word: A cache holds a group of neighboring words called *cache lines* (sometimes called *cache blocks*).

In practice, most processors have two or three levels of caches, called $L1$, $L2$, and $L3$ caches. All but the last (and largest) cache typically reside on the same chip as the processor. An L1 cache typically takes one or two cycles to access. An on-chip L2 may take about 10 cycles to access. The last level cache, whether L2 or L3, typically takes tens of cycles to access. These caches are significantly faster than the hundreds of cycles required to access the memory. Of course, these times vary from platform to platform, and many multiprocessors have even more elaborate cache structures.

The original proposals for NUMA architectures did not include caches because it was felt that local memory was enough. Later, however, commercial NUMA architectures did include caches. Sometimes the term *cache-coherent NUMA* (cc-NUMA) is used to mean NUMA architectures with caches. Here, to avoid ambiguity, we use NUMA to include cache-coherence unless we explicitly state otherwise.

Caches are expensive to build and therefore significantly smaller than the memory: Only a fraction of the memory locations will fit in a cache at the same time. We would therefore like the cache to maintain values of the most highly used locations. This implies that when a location needs to be cached and the cache is full, it is necessary to *evict* a line, discarding it if it has not been modified, and writing it back to main memory if it has. A *replacement policy* determines which cache line to replace to make room for a given new location. If the replacement policy is free to replace any line, then we say the cache is *fully associative*. If, on the other hand, there is only one line that can be replaced, then we say the cache is *direct-mapped*. If we split the difference, allowing any line from a *set* of size k to be replaced to make room for a given line, then we say the cache is *k-way set associative*.

B.5.1 Coherence

Sharing (or, less politely, *memory contention*) occurs when one processor reads or writes a memory address that is cached by another. If both processors are reading the data without modifying it, then the data can be cached at both processors. If, however, one processor tries to update the shared cache line, then the other's copy must be *invalidated* to ensure that it does not read an out-of-date value. In its most general form, this problem is called *cache-coherence*. The literature contains a variety of very complex and clever cache coherence protocols. Here we review one of the most commonly used, called the *MESI* protocol (pronounced "messy") after the names of possible cache line states. (Modern processors tend to use more complex protocols with additional states.) Here are the cache line states:

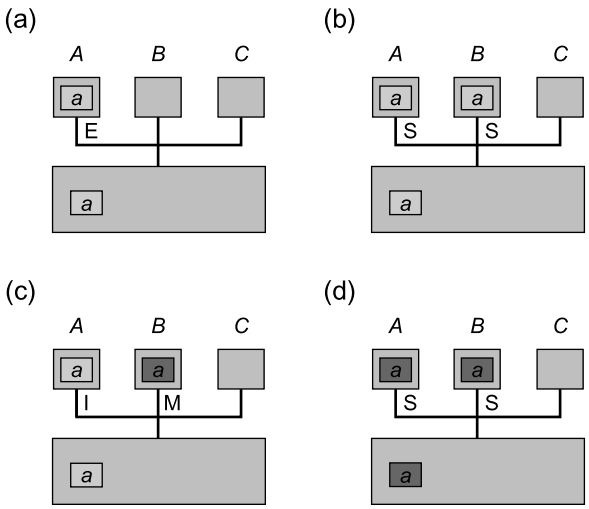

FIGURE B.5

Example of the MESI cache-coherence protocol's state transitions. (a) Processor *A* reads data from address *a*, and stores the data in its cache in the *exclusive* state. (b) When processor *B* attempts to read from the same address, *A* detects the address conflict, and responds with the associated data. Now *a* is cached at both *A* and *B* in the *shared* state. (c) If *B* writes to the shared address *a*, it changes its state to *modified*, and broadcasts a message warning *A* (and any other processor that might have those data cached) to set its cache line state to *invalid*. (d) If *A* then reads from *a*, it broadcasts a request, and *B* responds by sending the modified data both to *A* and to the main memory, leaving both copies in the *shared* state.

- *Modified*: The line has been modified in the cache, and it must eventually be written back to main memory. No other processor has this line cached.
- *Exclusive*: The line has not been modified, and no other processor has this line cached.
- *Shared*: The line has not been modified, and other processors may have this line cached.
- *Invalid*: The line does not contain meaningful data.

We illustrate this protocol by a short example depicted in Fig. B.5. For simplicity, we assume processors and memory are linked by a bus.

Processor *A* reads data from address *a*, and stores the data in its cache in the *exclusive* state. When processor *B* attempts to read from the same address, *A* detects the address conflict, and responds with the associated data. Now *a* is cached at both *A* and *B* in the *shared* state. If *B* writes to the shared address *a*, it changes its state to *modified*, and broadcasts a message warning *A* (and any other processor that might have those data cached) to set its cache line state to *invalid*. If *A* then reads from *a*, it broadcasts a request, and *B* responds by sending the modified data both to *A* and to the main memory, leaving both copies in the *shared* state.

False sharing occurs when processors that are accessing logically distinct data nevertheless conflict because the locations they are accessing lie on the same cache line. This observation illustrates a difficult trade-off: Large cache lines are good for locality, but they increase the likelihood of false sharing. The likelihood of false sharing can be reduced by ensuring that data objects that might be accessed concurrently by independent threads lie far enough apart in memory. For example, having multiple threads share a byte array invites false sharing, but having them share an array of double-precision integers is less dangerous.

B.5.2 Spinning

A processor is *spinning* if it is repeatedly testing some word in memory, waiting for another processor to change it. Depending on the architecture, spinning can have a dramatic effect on overall system performance.

On an SMP architecture without caches, spinning is a very bad idea. Each time the processor reads the memory, it consumes bus bandwidth without accomplishing any useful work. Because the bus is a broadcast medium, these requests directed to memory may prevent other processors from making progress.

On a NUMA architecture without caches, spinning may be acceptable if the address in question resides in the processor's local memory. Even though multiprocessor architectures without caches are rare, we will still ask, when we consider a synchronization protocol that involves spinning, whether it permits each processor to spin on its own local memory.

On an SMP or NUMA architecture with caches, spinning consumes significantly fewer resources. The first time the processor reads the address, it takes a cache miss, and loads the contents of that address into a cache line. Thereafter, as long as those data remain unchanged, the processor simply rereads from its own cache, consuming no interconnect bandwidth, a process known as *local spinning*. When the cache state changes, the processor takes a single cache miss, observes that the data have changed, and stops spinning.

B.6 Cache-conscious programming, or the puzzle solved

We now know enough to explain why the `TTASLock` examined in Appendix B.1 outperforms the `TASLock`. Each time the `TASLock` applies `getAndSet(`*true*`)` to the lock, it sends a message on the interconnect causing a substantial amount of traffic. In an SMP architecture, the resulting traffic may be enough to saturate the interconnect, delaying all threads, including a thread trying to release the lock, or even threads not contending for the lock. By contrast, while the lock is busy, the `TTASLock` spins, reading a locally cached copy of the lock, and producing no interconnect traffic, explaining its improved performance.

The TTASLock is still far from ideal. When the lock is released, all its cached copies are invalidated, and all waiting threads call getAndSet(*true*), resulting in a burst of traffic, smaller than that of the TASLock, but nevertheless significant.

We further discuss the interactions of caches with locking in Chapter 7. Here we consider some simple ways to structure data to avoid false sharing.

- Objects or fields that are accessed independently should be aligned and padded so that they end up on different cache lines.
- Keep read-only data separate from data that are modified frequently. For example, consider a list whose structure is constant, but whose elements' value fields change frequently. To ensure that modifications do not slow down list traversals, one could align and pad the value fields so that each one fills up a cache line.
- When possible, split an object into thread-local pieces. For example, a counter used for statistics could be split into an array of counters, one per thread, each one residing on a different cache line. Splitting the counter allows each thread to update its own replica, avoiding the invalidation traffic that would be incurred by having a single shared counter.
- If a lock protects data that is frequently modified, then keep the lock and the data on distinct cache lines, so that threads trying to acquire the lock do not interfere with the lock-holder's access to the data.
- If a lock protects data that are frequently uncontended, then try to keep the lock and the data on the same cache lines, so that acquiring the lock will also load some of the data into the cache.

B.7 Multicore and multithreaded architectures

In a *multicore* architecture, as in Fig. B.6, multiple processors are placed on the same chip. Each processor on that chip typically has its own L1 cache, but they share a

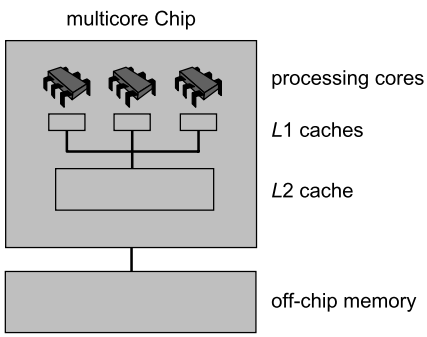

FIGURE B.6

A multicore SMP architecture. The *L2* cache is on chip and shared by all processors while the memory is off-chip.

common L2 cache. Processors can communicate efficiently through the shared L2 cache, avoiding the need to go through memory, and to invoke the cumbersome cache-coherence protocol.

In a *multithreaded* architecture, a single processor may execute two or more threads at once. Many modern processors have substantial internal parallelism. They can execute instructions out of order, or in parallel (e.g., keeping both fixed and floating-point units busy), or even execute instructions speculatively before branches or data have been computed. To keep hardware units busy, multithreaded processors can mix instructions from multiple streams.

Modern processor architectures combine multicore with multithreading, where multiple individually multithreaded cores may reside on the same chip. The context switches on some multicore chips are inexpensive and are performed at a very fine granularity, essentially context switching on every instruction. Thus, multithreading serves to hide the high latency of accessing memory: Whenever a thread accesses memory, the processor allows another thread to execute.

B.7.1 Relaxed memory consistency

When a processor writes a value to memory, that value is kept in the cache and marked as *dirty*, meaning that it must eventually be written back to main memory. On most modern processors, write requests are not applied to memory when they are issued. Rather, they are collected in a hardware queue called a *write buffer* (or *store buffer*), and applied to memory together at a later time. A write buffer provides two benefits. First, it is often more efficient to issue several requests together, a phenomenon called *batching*. Second, if a thread writes to an address more than once, earlier requests can be discarded, saving a trip to memory, a phenomenon called *write absorption*.

The use of write buffers has a very important consequence: The order in which reads and writes are issued to memory is not necessarily the order in which they occur in the program. For example, recall the flag principle of Chapter 1, which was crucial to the correctness of mutual exclusion: If two processors each first write their own flag and then read the other's flag location, then one of them will see the other's newly written flag value. With write buffers, this is no longer true: both threads may write, each in its respective write buffer, but these writes might not be applied to the shared memory until after each processor reads the other's flag location in memory. Thus, neither reads the other's flag.

Compilers make matters even worse. They are good at optimizing performance on single-processor architectures. Often, this optimization involves *reordering* a thread's reads and writes to memory. Such reordering is invisible for single-threaded programs, but it can have unexpected consequences for multithreaded programs in which threads may observe the order in which writes occur. For example, if one thread fills a buffer with data and then sets an indicator to mark the buffer as full, then concurrent threads may see the indicator set before they see the new data, causing them to read stale values. The erroneous *double-checked locking* pattern described in Appendix A.3 is an example of a pitfall produced by unintuitive aspects of the Java memory model.

Different architectures provide different guarantees about the extent to which memory reads and writes can be reordered. As a rule, it is better not to rely on such guarantees, and instead to use more expensive techniques, described in the following paragraph, to prevent such reordering.

Every architecture provides a *memory barrier* instruction (sometimes called a *fence*), which forces writes to take place in the order they are issued, but at a price. A memory barrier flushes the write buffer, ensuring that all writes issued before the barrier become visible to the processor that issued the barrier. Memory barriers are often inserted automatically by atomic read–modify–write operations such as `getAndSet()`, or by standard concurrency libraries. Thus, explicit use of memory barriers is needed only when processors perform read–write instructions on shared variables outside of critical sections.

On one hand, memory barriers are expensive (hundreds of cycles, maybe more), and should be used only when necessary. On the other hand, synchronization bugs can be very difficult to track down, so memory barriers should be used liberally, rather than relying on complex platform-specific guarantees about limits to memory instruction reordering.

The Java language itself allows reads and writes to object fields to be reordered if they occur outside **`synchronized`** methods or blocks. Java provides a **`volatile`** keyword that ensures that reads and writes to a **`volatile`** object field that occur outside **`synchronized`** blocks or methods are not reordered. (The `atomic` template provides similar guarantees for C++.) Using this keyword can be expensive, so it should be used only when necessary. We note that in principle, one could use volatile fields to make double-checked locking work correctly, but there would not be much point, since accessing volatile variables requires synchronization anyway.

Here ends our primer on multiprocessor hardware. We now continue to discuss these architectural concepts in the context of specific data structures and algorithms. A pattern will emerge: The performance of multiprocessor programs is highly dependent on synergy with the underlying hardware.

B.8 Hardware synchronization instructions

As discussed in Chapter 5, any modern multiprocessor architecture must support powerful synchronization primitives to be universal, that is, to provide concurrent computation's equivalent of a universal Turing machine. It is therefore not surprising that implementations of Java and C++ rely on such specialized hardware instructions (also called hardware primitives) in implementing synchronization, from spin locks and monitors to the most complex lock-free data structures.

Modern architectures typically provide one of two kinds of universal synchronization primitives. The *compare-and-swap* (CAS) instruction is supported in architectures by AMD, Intel, and Oracle. It takes three arguments: an address a in memory, an *expected* value e, and an *update* value v, and returns a Boolean. It *atomically* executes the following: If the memory at address a contains the expected value e, write