Synchronization

A treatment of parallel programming warrants at least a cursory mention of the vast topic of synchronization. In the simple examples considered throughout this text, we have seen numerous cases of multiple threads accessing a shared memory location, be it a complex collection or a single integer. Aside from read-only data, every access to a shared memory location requires synchronization, but not all synchronization mechanisms have the same performance and scalability costs.

Before we begin, let's revisit the need for synchronization when accessing small amounts of data. Modern CPUs can issue atomic reads and writes to memory; for example, a write of a 32 bit integer always executes atomically. This means that if one processor writes the value `0xDEADBEEF` to a memory location previously initialized with the value 0, another processor will not observe the memory location with a partial update, such as `0xDEAD0000` or `0x0000BEEF`. Unfortunately, the same thing is not true of larger memory locations; for example, even on a 64 bit processor, writing 20 bytes into memory is not an atomic operation and cannot be performed atomically.

However, even when accessing a 32 bit memory location but issuing multiple operations, synchronization problems arise immediately. For example, the operation `++i` (where `i` is a stack variable of type `int`) is typically translated to a sequence of three machine instructions:

```
mov eax, dword ptr [ebp-64] ;copy from stack to register
inc eax ;increment value in register
mov dword ptr [ebp-64], eax ;copy from register to stack
```

Each of these instructions executes atomically, but without additional synchronization it is possible for two processors to execute parts of the instruction sequence concurrently, resulting in *lost updates*. Suppose that the variable's initial value was 100, and examine the following execution history:

```
Processor #1    Processor #2
```

```
mov eax, dword ptr [ebp-64]
    mov eax, dword ptr [ebp-64] inc eax
inc eax
mov dword ptr [ebp-64], eax
    mov dword ptr [ebp-64], eax
```

In this case, the variable's eventual value will be 101, even though *two* processors have executed the increment operation and should have brought it to 102. This race condition—which is hopefully obvious and easily detectable—is a representative example of the situations that warrant careful synchronization.

OTHER DIRECTIONS

Many researchers and programming language designers do not believe that the situation governing shared memory synchronization can be addressed without changing completely the semantics of programming languages, parallelism frameworks, or processor memory models. There are several interesting directions in this area:

- Transactional memory in hardware or software suggests an explicit or implicit isolation model around memory operations and rollback semantics for series of memory operations. Currently, the performance cost of such approaches impedes their wide adoption in mainstream programming languages and frameworks.
- Agent-based languages bake a concurrency model deep into the language and require explicit communication between agents (objects) in terms of message-passing instead of shared memory access.
- Message-passing processor and memory architectures organize the system using a private-memory paradigm, where access to a shared memory location must be explicit through message-passing at the hardware level.

Throughout the rest of this section, we shall assume a more pragmatic view and attempt to reconcile the problems of shared memory synchronization by offering a set of synchronization mechanisms and patterns. However, the authors firmly believe that synchronization is more difficult than it should be; our shared experience demonstrates that a large majority of difficult bugs in software today stem from the simplicity of corrupting shared state by improperly synchronizing parallel programs. We hope that in a few years—or decades—the computing community will come up with somewhat better alternatives.

Lock-Free Code

One approach to synchronization places the burden on the operating system. After all, the operating system provides the facilities for creating and managing threads, and assumes full responsibility for scheduling their execution. It is then natural to expect from it to provide a set of synchronization primitives. Although we will discuss Windows synchronization mechanisms shortly, this approach begs the question of how the operating system *implements* these synchronization mechanisms. Surely Windows itself is in need of synchronizing access to its internal data structures—even the data structures representing other synchronization mechanisms—and it cannot implement synchronization mechanisms by deferring to them recursively. It also turns out that Windows synchronization mechanisms often require a system call (user-mode to kernel-mode transition) and thread context switch to ensure synchronization, which is relatively expensive if the operations that require synchronization are very cheap (such as incrementing a number or inserting an item into a linked list).

All the processor families on which Windows can run implement a *hardware* synchronization primitive called Compare-And-Swap (CAS). CAS has the following semantics (in pseudo-code), and executes *atomically*:

```
WORD CAS(WORD* location, WORD value, WORD comparand) {
  WORD old = *location;
  if (old == comparand) {
    *location = value;
  }
  return old;
}
```

Simply put, CAS compares a memory location with a provided value. If the memory location contains the provided value, it is replaced by another value; otherwise, it is unchanged. In any case, the content of the memory location prior to the operation is returned.

For example, on Intel x86 processors, the `LOCK CMPXCHG` instruction implements this primitive. Translating a `CAS(&a,b,c)` call to `LOCK CMPXCHG` is a simple mechanical process, which is why we will be content with using CAS throughout the rest of this section. In the .NET Framework, CAS is implemented using a set of overloads called `Interlocked.CompareExchange`.

```
//C# code:
int n = . . .;
```

```
if (Interlocked.CompareExchange(ref n, 1, 0) == 0) { //attempt to replace 0 with 1
  //. . .do something
}
//x86 assembly instructions:
mov eax, 0 ;the comparand
mov edx, 1 ;the new value
lock cmpxchg dword ptr [ebp-64], edx ;assume that n is in [ebp-64]
test eax, eax ;if eax = 0, the replace took place
jnz not_taken
;. . .do something
not_taken:
```

A single CAS operation is often not enough to ensure any useful synchronization, unless the desirable semantics are to perform a one-time check-and-replace operation. However, when combined with a looping construct, CAS can be used for a non-negligible variety of synchronization tasks. First, we consider a simple example of in-place multiplication. We want to execute the operation `x *= y` atomically, where `x` is a shared memory location that may be written to simultaneously by other threads, and `y` is a constant value that is not modified by other threads. The following CAS-based C# method performs this task:

```
public static void InterlockedMultiplyInPlace(ref int x, int y) {
  int temp, mult;
  do {
    temp = x;
    mult = temp * y;
  } while(Interlocked.CompareExchange(ref x, mult, temp) ! = temp);
}
```

Each loop iteration begins by reading the value of `x` to a temporary stack variable, which cannot be modified by another thread. Next, we find the multiplication result, ready to be placed into `x`. Finally, the loop terminates if and only if `CompareExchange` reports that it successfully replaced the value of `x` with the multiplication result, granted that the original value was not modified. We cannot guarantee that the loop will terminate in a bounded number of iterations; however, it is highly unlikely that—even under pressure from other processors—a single processor will be skipped more than a few times when trying to replace `x` with its new value. Nonetheless, the loop must be prepared to face this case (and try again). Consider the following execution history with `x = 3`, `y = 5` on two processors:

```
Processor #1    Processor #2

temp = x; (3)
    temp = x; (3)
mult = temp * y; (15)
    mult = temp * y; (15)
    CAS(ref x, mult, temp) == 3 (== temp)
CAS(ref x, mult, temp) == 15 (! = temp)
```

Even this extremely simple example is very easy to get wrong. For example, the following loop may cause lost updates:

```
public static void InterlockedMultiplyInPlace(ref int x, int y) {
  int temp, mult;
  do {
    temp = x;
    mult = x * y;
  } while(Interlocked.CompareExchange(ref x, mult, temp) ! = temp);
}
```

Why? Reading the value of `x` twice in rapid succession does not guarantee that we see the same value! The following execution history demonstrates how an incorrect result can be produced, with `x = 3`, `y = 5` on two processors—at the end of the execution `x = 60`!

```
Processor #1    Processor #2

temp = x; (3)
    x = 12;
mult = x * y; (60!)
    x = 3;
CAS(ref x, mult, temp) == 3 (== temp)
```

We can generalize this result to any algorithm that needs to read only a single mutating memory location and replace it with a new value, no matter how complex. The most general version would be the following:

```
public static void DoWithCAS < T > (ref T location, Func < T,T > generator) where T : class {
  T temp, replace;
  do {
    temp = location;
    replace = generator(temp);
  } while (Interlocked.CompareExchange(ref location, replace, temp) ! = temp);
}
```

Expressing the multiplication method in terms of this general version is very easy:

```
public static void InterlockedMultiplyInPlace(ref int x, int y) {
  DoWithCAS(ref x, t => t * y);
}
```

Specifically, there is a simple synchronization mechanism called *spinlock* that can be implemented using CAS. The idea here is as follows: to acquire a lock is to make sure that any other thread that attempts to acquire it will fail and try again. A spinlock, then, is a lock that allows a single thread to acquire it and all other threads to *spin* ("waste" CPU cycles) while trying to acquire it:

```
public class SpinLock {
  private volatile int locked;
  public void Acquire() {
    while (Interlocked.CompareExchange(ref locked, 1, 0) ! = 0);
  }
  public void Release() {
    locked = 0;
  }
}
```

MEMORY MODELS AND VOLATILE VARIABLES

A complete treatment of synchronization would include a discussion of memory models and the need for volatile variables. However, we lack the space to cover this subject adequately, and offer only a brief account. Joe Duffy's book "Concurrent Programming on Windows" (Addison-Wesley, 2008) offers an in-depth detailed description.

Generally speaking, a memory model for a particular language/environment describes how the compiler and the processor hardware may reorder operations on memory performed by different threads—the interaction of threads through shared memory. Although most memory models agree that read and write operations on the *same* memory location may not be reordered, there is scarce agreement on the semantics of read and write operations on *different* memory locations. For example, the following program may output 13 when starting from the state `f = 0, x = 13`:

```
Processor #1     Processor #2

while (f == 0);     x = 42;
print(x);     f = 1;
```

The reason for this unintuitive result is that the compiler and processor are free to reorder the instructions on processor #2 such that the write to `f` completes before the write to `x`, and to reorder the instructions on processor #1 such that the read of `x` completes before the read of `f`. Failing to take into account the details of a particular memory model may lead to extremely difficult bugs.

There are several remedies available to C# developers when dealing with memory reordering issues. First is the `volatile` keyword, which prevents compiler reorderings and most processor reorderings around operations on a particular variable. Second is the set of `Interlocked` APIs and `Thread.MemoryBarrier`, which introduce a fence which cannot be crossed in one or both directions as far as reorderings are concerned. Fortunately, the Windows synchronization mechanisms (which involve a system call) as well as any lock-free synchronization primitives in the TPL issue a memory barrier when necessary. However, if you attempt the already-risky task of implementing your own low-level synchronization, you should invest a significant amount of time understanding the details of your target environment's memory model.

We cannot stress this harder: if you choose to deal with memory ordering directly, it is *absolutely crucial* that you understand the memory model of every language and hardware combination you use for programming your multithreaded applications. There will be no framework to guard your steps.

In our spinlock implementation, 0 represents a free lock and 1 represents a lock that is taken. Our implementation attempts to replace its internal value with 1, provided that its current value is 0—i.e., acquire the lock, provided that it is not currently acquired. Because there is no guarantee that the owning thread will release the lock quickly, using a spinlock means that you may devote a set of threads to spinning around, wasting CPU cycles, waiting for a lock to become available. This makes spinlocks inapplicable for protecting operations such as database access, writing out a large file to disk, sending a packet over the network, and similar long-running operations. However, spinlocks are very useful when the guarded code section is very quick—modifying a bunch of fields on an object, incrementing several variables in a row, or inserting an item into a simple collection.

Indeed, the Windows kernel itself uses spinlocks extensively to implement internal synchronization. Kernel data structures such as the scheduler database, file system cache block list, memory page frame number database and others are protected by one or more spinlocks. Moreover, the Windows kernel introduces additional optimizations to the simple spinlock implementation described above, which suffers from two problems:

1. The spinlock is not *fair*, in terms of FIFO semantics. A processor may be the last of ten processors to call the `Acquire` method and spin inside it, but may be the first to actually acquire it after it has been released by its owner.
2. When the spinlock owner releases the spinlock, it invalidates the cache of all the processors currently spinning in the `Acquire` method, although only one processor will actually acquire it. (We will revisit cache invalidation later in this chapter.)

The Windows kernel uses *in-stack queued spinlocks*; an in-stack queued spinlock maintains a queue of processors waiting for a lock, and every processor waiting for the lock spins around a separate memory location, which is not in the cache of other processors. When the spinlock's owner releases the lock, it finds the first processor in the queue and signals the bit on which this particular processor is waiting. This guarantees FIFO semantics and prevents cache invalidations on all processors but the one that successfully acquires the lock.

Note Production-grade implementations of spinlocks can be more robust in face of failures, avoid spinning for more than a reasonable threshold (by converting spinning to a blocking wait), track the owning thread to make sure spinlocks are correctly acquired and released, allow recursive acquisition of locks, and provide additional facilities. The `SpinLock` type in the Task Parallel Library is one recommended implementation.

Armed with the CAS synchronization primitive, we now reach an incredible feat of engineering—a lock-free stack. In Chapter 5 we have considered some concurrent collections, and will not repeat this discussion, but the implementation of `ConcurrentStack < T >` remained somewhat of a mystery. Almost magically, `ConcurrentStack < T >` allows multiple threads to push and pop items from it, but never requires a blocking synchronization mechanism (that we consider next) to do so.

We shall implement a lock-free stack by using a singly linked list. The stack's top element is the head of the list; pushing an item onto the stack or popping an item from the stack means replacing the head of the list. To do this in a synchronized fashion, we rely on the CAS primitive; in fact, we can use the `DoWithCAS < T>` helper introduced previously:

```
public class LockFreeStack < T > {
  private class Node {
    public T Data;
    public Node Next;
  }
  private Node head;
  public void Push(T element) {
    Node node = new Node { Data = element };
    DoWithCAS(ref head, h => {
    node.Next = h;
    return node;
    });
  }
  public bool TryPop(out T element) {
    //DoWithCAS does not work here because we need early termination semantics
    Node node;
    do {
    node = head;
    if (node == null) {
    element = default(T);
    return false; //bail out – nothing to return
    }
    } while (Interlocked.CompareExchange(ref head, node.Next, node) ! = node);
    element = node.Data;
    return true;
  }
}
```

The `Push` method attempts to replace the list head with a new node, whose `Next` pointer points to the current list head. Similarly, the `TryPop` method attempts to replace the list head with the node to which the current head's `Next` pointer points, as illustrated in Figure 6-8.

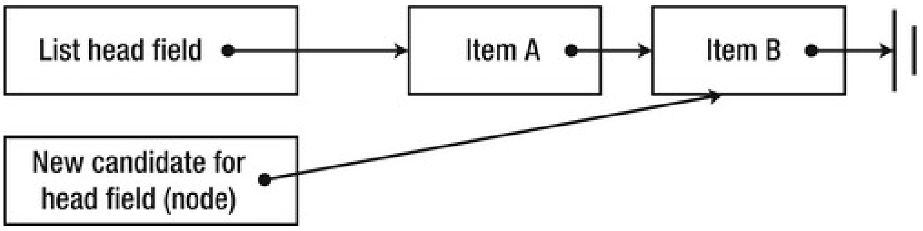

Figure 6-8. The TryPop operation attempts to replace the current list head with a new one

You may be tempted to think that every data structure in the world can be implemented using CAS and similar lock-free primitives. Indeed, there are some additional examples of lock-free collections in wide use today:

- Lock-free doubly linked list
- Lock-free queue (with a head and tail)
- Lock-free simple priority queue

However, there is a great variety of collections that cannot be easily implemented using lock-free code, and still rely on blocking synchronization mechanisms. Furthermore, there is a considerable amount of code that requires synchronization but cannot use CAS because it takes too long to execute. We now turn to discuss "real" synchronization mechanisms, which involve blocking, implemented by the operating system.

Windows Synchronization Mechanisms

Windows offers numerous synchronization mechanisms to user-mode programs, such as events, semaphores, mutexes, and condition variables. Our programs can access these synchronization mechanisms through handles and Win32 API calls, which issue the corresponding system calls on our behalf. The .NET Framework wraps most Windows synchronization mechanisms in thin object-oriented packages, such as `ManualResetEvent`, `Mutex`, `Semaphore`, and others. On top of the existing synchronization mechanisms, .NET offers several new ones, such as `ReaderWriterLockSlim` and `Monitor`. We will not examine exhaustively every synchronization mechanism in minute detail, which is a task best left to API documentation; it is important, however, to understand their general performance characteristics.

The Windows kernel implements the synchronization mechanisms we are now discussing by blocking a thread that attempts to acquire the lock when the lock is not available. Blocking a thread involves removing it from the CPU, marking it as waiting, and scheduling another thread for execution. This operation involves a system call, which is a user-mode to kernel-mode transition, a context switch between two threads, and a small set of data structure updates (see Figure 6-9) performed in the kernel to mark the thread as waiting and associate it with the synchronization mechanism for which it's waiting.

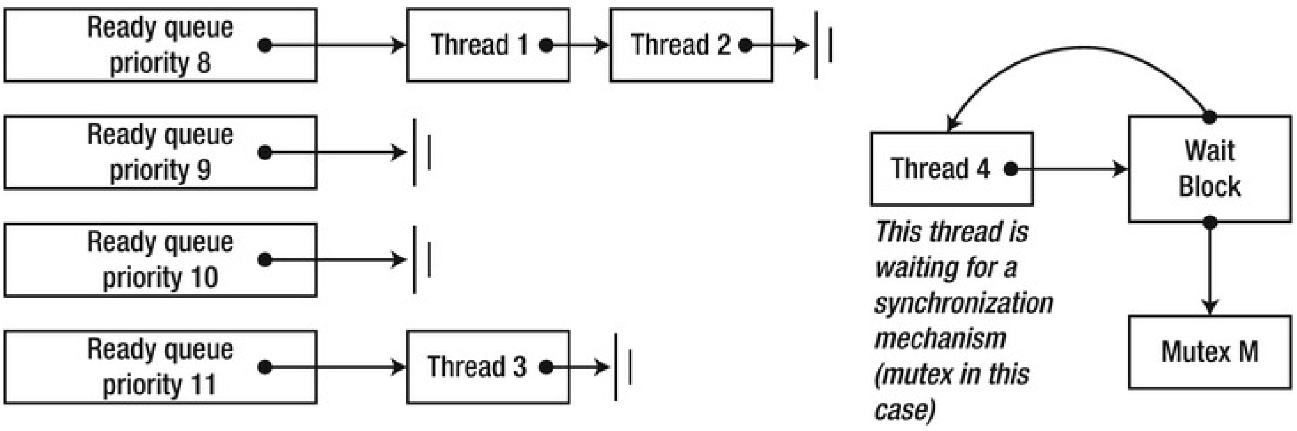

Figure 6-9 . The data maintained by the operating system scheduler. Threads ready for execution are placed in FIFO queues, sorted by priority. Blocked threads reference their synchronization mechanism through an internal structure called a wait block

Overall, there are potentially thousands of CPU cycles spent to block a thread, and a similar number of cycles is required to unblock it when the synchronization mechanism becomes available. It is clear, then, that if a kernel synchronization mechanism is used to protect a long-running operation, such as writing a large buffer to a file or performing a network round-trip, this overhead is negligible, but if a kernel synchronization mechanism is used to protect an operation like `++i`, this overhead introduces an inexcusable slowdown.

The synchronization mechanisms Windows and .NET make available to applications differ primarily in terms of their acquire and release semantics, also known as their signal state. When a synchronization mechanism becomes signaled, it wakes up a thread (or a group of threads) waiting for it to become available. Below are the signal state semantics for some of the synchronization mechanisms currently accessible to .NET applications:

Table 6-1. *Signal State Semantics of Some Synchronization Mechanisms*

Synchronization Mechanism	When Does It Become Signaled?	Which Threads Are Woken?
`Mutex`	When a thread calls `Mutex.ReleaseMutex`	One of the threads that are waiting for the mutex
`Semaphore`	When a thread calls `Semaphore.Release`	One of the threads that are waiting for the semaphore
`ManualResetEvent`	When a thread calls `ManualResetEvent.Set`	All of the threads that are waiting for the event
`AutoResetEvent`	When a thread calls `AutoResetEvent.Set`	One of the threads that are waiting for the event
`Monitor`	When a thread calls `Monitor.Exit`	One of the threads that are waiting for the Monitor
`Barrier`	When all the participating threads have called `Barrier.SignalAndWait`	All of the threads that are waiting for the barrier
`ReaderWriterLock`—for reading	When there are no writer threads, or the last writer thread has released the lock for writing	All of the threads that are waiting to enter the lock for reading
`ReaderWriterLock`—for writing	When there are no reader threads or writer threads	One of the threads that are waiting to enter the lock for writing

Other than the signal state semantics, some synchronization mechanisms differ also in terms of their internal implementation. For example, the Win32 critical section and the CLR Monitor implement an optimization for locks that are currently available. With that optimization, a thread attempting to acquire an available lock can grab it directly without performing a system call. On a different front, the reader-writer lock family of synchronization mechanisms distinguishes between readers and writers accessing a certain object, which permits better scalability when the data is most often accessed for reading.

Choosing the appropriate synchronization mechanism from the list of what Windows and .NET have to offer is often difficult, and there are times when a custom synchronization mechanism may offer better performance characteristics or more convenient semantics than the existing ones. We will not consider synchronization mechanisms any further; it is your responsibility when programming concurrent applications to choose responsibly between lock-free synchronization primitives and blocking synchronization mechanisms, and to determine the best combination of synchronization mechanisms to use.

Note No discussion of synchronization could be complete without highlighting data structures (collections) designed from the ground up for concurrency. Such collections are thread-safe—they allow safe access from multiple threads—as well as scalable without introducing unreasonable performance degradation due to locking. For a discussion of concurrent collections, as well as designing concurrent collections, consult Chapter 5.

Cache Considerations

We have previously paid a visit to the subject of processor caches in the context of collection implementation and memory density. In parallel programs it is similarly important to regard cache size and hit rates on a single processor, but it is even more important to consider how the caches of multiple processors interact. We will now consider a single representative example, which demonstrates the important of cache-oriented optimization, and emphasizes the value of good tools when it concerns performance optimization in general.

First, examine the following sequential method. It performs the rudimentary task of summing all the elements in a two-dimensional array of integers and returns the result.

```
public static int MatrixSumSequential(int[,] matrix) {
  int sum = 0;
  int rows = matrix.GetUpperBound(0);
  int cols = matrix.GetUpperBound(1);
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
    sum + = matrix[i,j];
    }
  }
  return sum;
}
```

We have in our arsenal a large set of tools for parallelizing programs of this sort. However, imagine for a moment that we don't have the TPL at our disposal, and choose to work directly with threads instead. The following attempt at parallelization may appear sufficiently reasonable to harvest the fruits of multi-core execution, and even implements a crude aggregation to avoid synchronization on the shared `sum` variable:

```
public static int MatrixSumParallel(int[,] matrix) {
  int sum = 0;
  int rows = matrix.GetUpperBound(0);
  int cols = matrix.GetUpperBound(1);
```

```
    const int THREADS = 4;
    int chunk = rows/THREADS; //should divide evenly
    int[] localSums = new int[THREADS];
    Thread[] threads = new Thread[THREADS];
    for (int i = 0; i < THREADS; ++i) {
      int start = chunk*i;
      int end = chunk*(i + 1);
      int threadNum = i; //prevent the compiler from hoisting the variable in the lambda captur
      threads[i] = new Thread(() => {
      for (int row = start; row < end; ++row) {
      for (int col = 0; col < cols; ++col) {
      localSums[threadNum] + = matrix[row,col];
      }
      }
      });
      threads[i].Start();
    }
    foreach (Thread thread in threads) {
      thread.Join();
    }
    sum = localSums.Sum();
    return sum;
  }
```

Executing each of the two methods 25 times on an Intel i7 processor produced the following results for a 2,000 × 2,000 matrix of integers: the sequential method completed within 325ms on average, whereas the parallelized method took a whopping 935ms on average, thrice as slow as the sequential version!

This is clearly unacceptable, but why? This is not another example of too fine-grained parallelism, because the number of threads is only 4. If you accept the premise that the problem is somehow cache-related (because this example appears in the "Cache Considerations" section), it would make sense to measure the number of cache misses introduced by the two methods. The Visual Studio profiler (when sampling at each 2,000 cache misses) reported 963 exclusive samples in the parallel version and only 659 exclusive samples in the sequential version; the vast majority of samples were on the inner loop line that reads from the matrix.

Again, why? Why would the line of code writing to the `localSums` array introduce so many more cache misses than the line writing to the `sum` local variable? The simple answer is that the writes to the shared array *invalidate cache lines at other processors*, causing every `+ =` operation on the array to be a cache miss.

As you recall from Chapter 5, processor caches are organized in cache lines, and adjacent memory locations share the same cache line. When one processor writes to a memory location that is in the cache of another processor, the hardware causes a cache invalidation that marks the cache line in the other processor's cache as invalid. Accessing an invalid cache line causes a cache miss. In our example above, it is very likely that the entire `localSums` array fits in a single cache line, and resides simultaneously in the caches of *all* four processors on which the application's threads are executing. Every write performed to any element of the array on any of the processors invalidates the cache line on all other processors, causing a constant ping-pong of cache invalidations (see Figure 6-10).

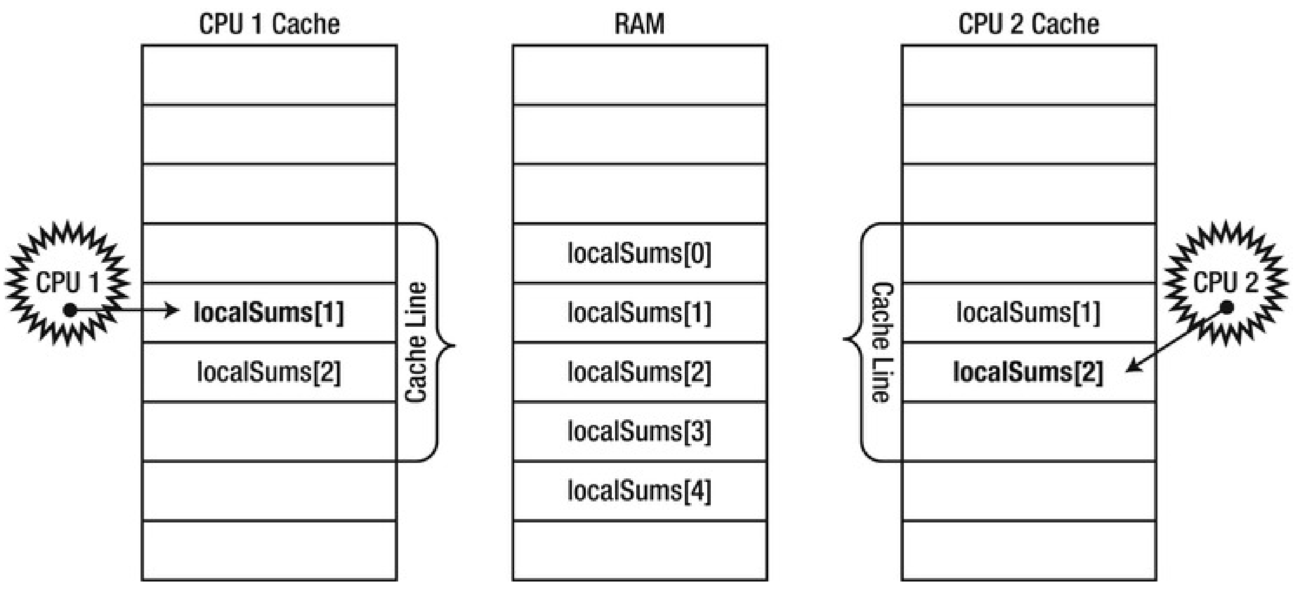

Figure 6-10. CPU 1 writes to localSums[1] while CPU 2 writes to localSums[2]. Because both array elements are adjacent and fit into the same cache line in both processor caches, each such write causes a cache invalidation on the other processor

To make sure that the problem is completely related to cache invalidations, it is possible to make the array strides sufficiently larger such that cache invalidation does not occur, or to replace the direct writes to the array with writes to a local variable in each thread that is eventually flushed to the array when the thread completes. Either of these optimizations restores sanity to the world, and makes the parallel version faster than the sequential one on a sufficiently large number of cores.

Cache invalidation (or cache collision) is a nasty problem that is exceptionally difficult to detect in a real application, even when aided by powerful profilers. Taking it into consideration upfront when designing CPU-bound algorithms will save you a lot of time and aggravation later.

Note The authors encountered a similar case of cache invalidation in a production scenario with a shared work item queue between two threads executing on two different processors. When a certain minor change was made to the internal structure of the queue class' fields, significant performance degradation (approximately 20%) was detected in subsequent builds. Upon very long and detailed examination, it became apparent that reordering fields in the queue class was responsible for the performance degradation; two fields that were being written to by different threads have become too close together and were placed on the same cache line. Adding padding betwee the fields restored the queue's performance to acceptable levels.