2.3.4 Cache coherence

Recall that CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems. To understand these issues, suppose we have a shared-memory system with two cores, each of which has its own private data cache. See Figure 2.17. As long as the two cores only read shared data, there is no problem. For example, suppose that `x` is a shared variable that has been initialized to 2, `y0` is private and owned by core 0, and `y1` and `z1` are private and owned by core 1. Now suppose the following statements are executed at the indicated times:

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving `x`
2	Statement(s) not involving `x`	`z1 = 4*x;`

Then the memory location for `y0` will eventually get the value 2, and the memory location for `y1` will eventually get the value 6. However, it's not so clear what value `z1` will get. It might at first appear that since core 0 updates `x` to 7 before the assignment to `z1`, `z1` will get the value $4 \times 7 = 28$. However, at time 0, `x` is in the cache of core 1. So unless for some reason `x` is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value `x = 2` may be used, and `z1` will get the value $4 \times 2 = 8$.

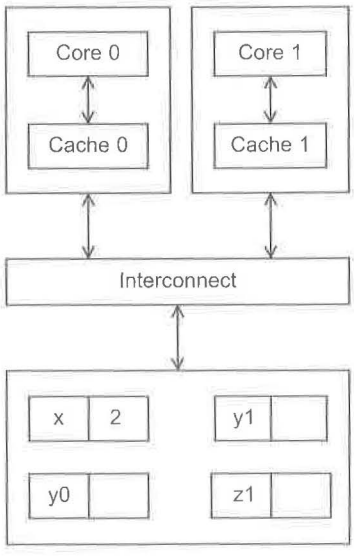

FIGURE 2.17

A shared-memory system with two cores and two caches

Note that this unpredictable behavior will occur regardless of whether the system is using a write-through or a write-back policy. If it's using a write-through policy, the main memory will be updated by the assignment `x = 7`. However, this will have no effect on the value in the cache of core 1. If the system is using a write-back policy, the new value of `x` in the cache of core 0 probably won't even be available to core 1 when it updates `z1`.

Clearly, this is a problem. The programmer doesn't have direct control over when the caches are updated, so her program cannot execute these apparently innocuous statements and know what will be stored in `z1`. There are several problems here, but the one we want to look at right now is that the caches we described for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the **cache coherence** problem.

Snooping cache coherence

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of `x` stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that `x` has been updated and it can mark its copy of `x` as invalid. This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the *cache line* containing `x` has been updated, not that `x` has been updated.

A couple of points should be made regarding snooping. First, it's not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated (but see Exercise 2.15). So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Figure 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable `x` stored in core 1's memory, by simply executing a statement such as `y = x`.

(Of course, accessing the memory attached to another core will be slower than accessing "local" memory, but that's another story.) Such a system can, in principle, scale to very large numbers of cores. However, snooping cache coherence is clearly a problem since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

False sharing

It's important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance. As an example, suppose we want to repeatedly call a function `f(i,j)` and add the computed values into a vector:

```
int i, j, m, n;
double y[m];

/* Assign y = 0 */
. . .

for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

We can parallelize this by dividing the iterations in the outer loop among the cores. If we have `core_count` cores, we might assign the first `m/core_count` iterations to the first core, the next `m/core_count` iterations to the second core, and so on.

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
```

```
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

. . .
```

Now suppose our shared-memory system has two cores, `m = 8`, doubles are eight bytes, cache lines are 64 bytes, and `y[0]` is stored at the beginning of a cache line. A cache line can store eight doubles, and `y` takes one full cache line. What happens when core 0 and core 1 simultaneously execute their codes? Since all of `y` is stored in a single cache line, each time one of the cores executes the statement `y[i] += f(i,j)`, the line will be invalidated, and the next time the other core tries to execute this statement it will have to fetch the updated line from memory! So if n is large, we would expect that a large percentage of the assignments `y[i] += f(i,j)` will access main memory—in spite of the fact that core 0 and core 1 never access each others' elements of `y`. This is called **false sharing**, because the system is behaving *as if* the elements of `y` were being shared by the cores.

Note that false sharing does not cause incorrect results. However, it can ruin the performance of a program by causing many more accesses to memory than necessary. We can reduce its effect by using temporary storage that is local to the thread or process and then copying the temporary storage to the shared storage. We'll return to the subject of false sharing in Chapters 4 and 5.

2.3.5 Shared-memory versus distributed-memory

Newcomers to parallel computing sometimes wonder why all MIMD systems aren't shared-memory, since most programmers find the concept of implicitly coordinating the work of the processors through shared data structures more appealing than explicitly sending messages. There are several issues, some of which we'll discuss when we talk about software for distributed- and shared-memory. However, the principal hardware issue is the cost of scaling the interconnect. As we add processors to a bus, the chance that there will be conflicts over access to the bus increase dramatically, so buses are suitable for systems with only a few processors. Large crossbars are very expensive, so it's also unusual to find systems with large crossbar interconnects. On the other hand, distributed-memory interconnects such as the hypercube and the toroidal mesh are relatively inexpensive, and distributed-memory systems with thousands of processors that use these and other interconnects have been built. Thus, distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.