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.

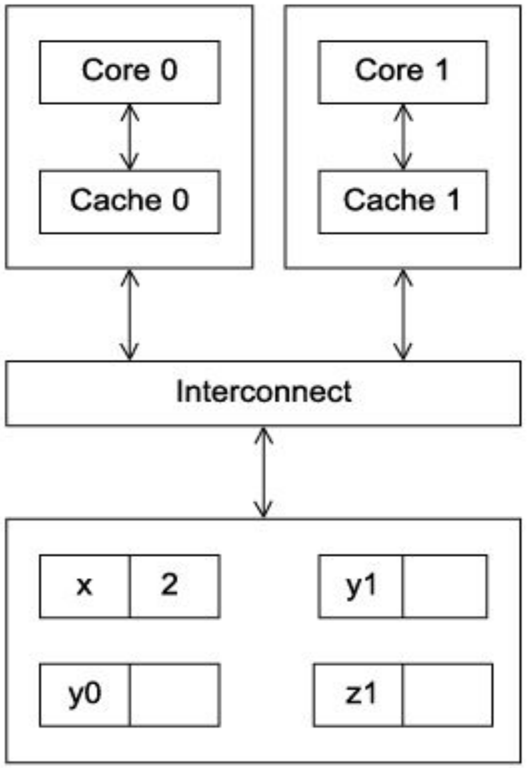
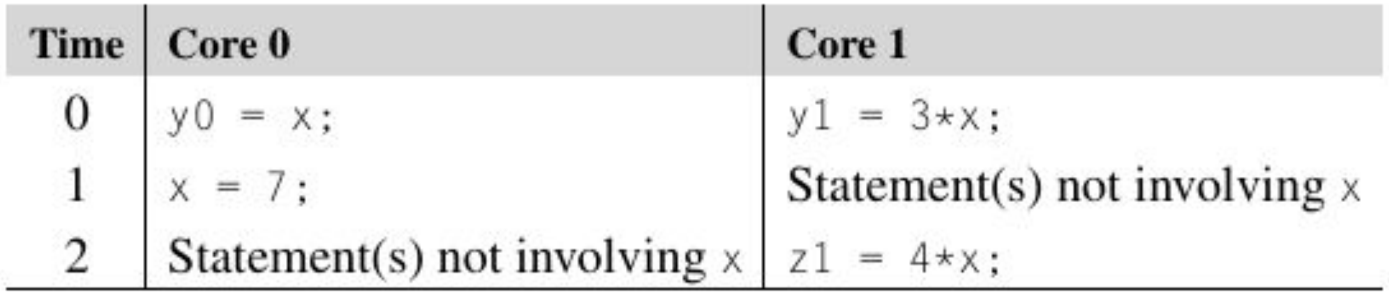


FIGURE 2.17

A shared-memory system with two cores and two caches.

- 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;`

- 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.

- When multiple processors (or cores) each have their own local cache but share a common main memory, copies of the same memory location may exist in more than one cache. **Cache coherence ensures that all processors see the most recent and correct value of that data**, regardless of where it is stored.

- There are two main approaches to ensuring cache coherence: **snooping cache coherence and directory-based cache coherence**.

- **Snooping 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. (**write-back caches & write-through**)

Directory-based cache coherence

- Unfortunately **i**n large networks, broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated. So snooping cache coherence isn’t scalable, because for larger systems it will cause performance to de grade.
- 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.
- 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 will invalidate those lines.

- **False sharing**
- 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];
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; 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.

- in spite of the fact that core 0 and core 1 never access each other's elements of y. This is called **false sharing**

Shared-memory vs Distributed-memory systems

Shared-memory systems are easier to program because processors share data structures directly. However, they do not scale well:

- Buses become congested as more processors are added.
- Crossbars can solve this but are too expensive for large systems.
- So, shared-memory is practical only for small to medium systems.

- **Distributed-memory systems** require programmers to explicitly send messages between processors, which is harder to program. But their interconnects (like hypercubes or toroidal meshes) are cheaper and scalable, allowing systems with thousands of processors to be built.

Coordinating the processes/threads

- In a very few cases, obtaining excellent parallel performance is trivial. For example, suppose we have two arrays and we want to add them:

```
double x[n] , y[n];
for ( int i = 0; i < n; i++)
x [i] += y[i];
```

- To parallelize this, we only need to assign elements of the arrays to the processes/threads.

1. Divide the work among the processes/threads in such a way that:

    a. each process/thread gets roughly the same amount of work,   b. the amount of communication required is minimized.

2. Arrange for the processes/threads to synchronize.

3. Arrange for communication among the processes/threads.

Shared-memory

- As we noted earlier, in shared-memory programs, variables can be shared or private.
- Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread.
- Communication among the threads is usually done through shared variables.

- Dynamic and static threads:
- In the dynamic threads paradigm, a master thread spawns worker threads only when new tasks arrive, and these workers terminate after completing their tasks.

- This ensures efficient resource utilization, as system resources are used only while threads are active. Such a model is well-suited for handling unpredictable or varying workloads efficiently.

- **static thread paradigm**, In this paradigm, all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may do some cleanup (e.g., free memory), and then it also terminates.

- Nondeterminism
- In any MIMD system in which the processors execute asynchronously it is likely that there will be nondeterminism.

- A computation is nondeterministic if a given input can result in different outputs.

- If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run.

- Suppose we have two threads, one with ID or rank 0 and the other with ID or rank 1. Suppose also that each is storing a private variable my_x, thread 0's value for my_x is 7, and thread 1's is 19.
- `printf("Thread %d > my_x = %d\n", my_rank , my_x);`
- Then the output could be

```
Thread 0 > my_x = 7
Thread 1 > my_x = 19
```

- but it could also be

```
Thread 1 > my_x = 19
Thread 0 > my_x = 7
```

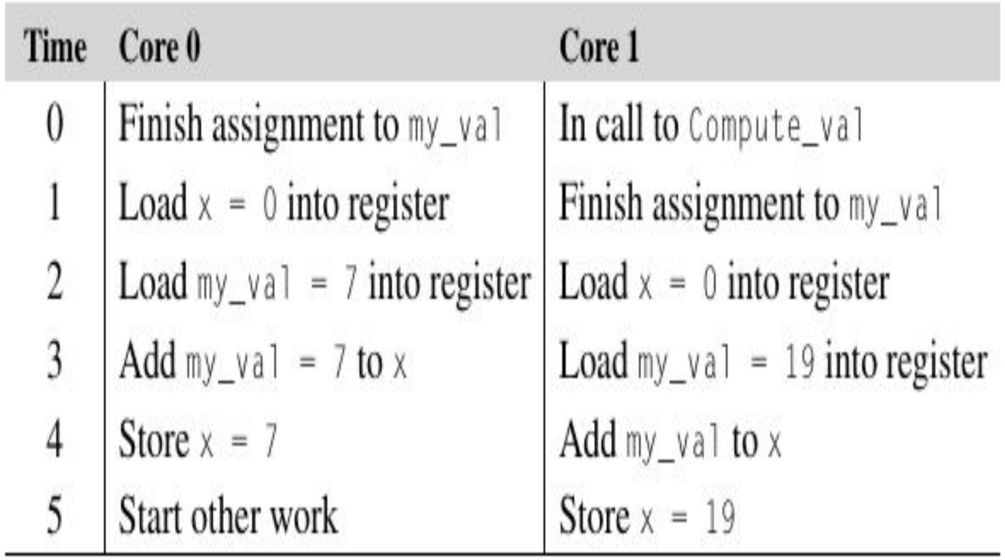
The time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete can't be predicted.

- There are also many cases in which nondeterminism— especially in shared-memory programs—can be disastrous, because it can easily result in program errors.

- Suppose each thread computes an int, which it stores in a private variable my_val. Suppose also that we want to add the values stored in my_val into a shared-memory location x that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
my_val = Compute_val(my_rank);
x += my_val;
```

- Race condition



Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19

- One possibility is to ensure that only one thread executes the update x += my_val at a time.

- A block of code that can only be executed by one thread at a time is called a critical section, and it's usually our job as programmers to ensure mutually exclusive access to a critical section.

- The most commonly used mechanism for ensuring mutual exclusion is a mutual exclusion lock or mutex, or simply lock.

- The basic idea is that each critical section is protected by a lock. Before a thread can execute the code in the critical section, it must "obtain" the mutex by calling a mutex function, and when it's done executing the code in the critical section, it should "relinquish" the mutex by calling an unlock function.

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

- There are alternatives to mutexes. In busy-waiting, a thread enters a loop, whose sole purpose is to test a condition.

- In our example, suppose there is a shared variable ok_for_1 that has been initialized to false. Then something like the following code can ensure that thread 1 won't update x until after thread 0 has updated it:

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);     /* Busy-wait loop */
x += my_val;              /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;       /* Let thread 1 update x */
```