CHAPTER

CACHE COHERENCY AND MEMORY CONSISTENCY

7

7.1 INTRODUCTION

Writing robust and efficient multithreaded programs requires a precise understanding of the behavior of memory system with respect to read and write operations from multiple processors. In a sequential program, it can be taken for granted that a read operation on a memory location will read the last value previously written at the same location. Indeed, in sequential codes the concept of *last write operation* is precisely defined by *program order*, namely, the order in which the read-write operations appear in the program. In a multiprocessor system threads are asynchronous, and read and write operations executed by different threads are not naturally related by program order. Therefore, the question of *when* a data value written to a memory location by one thread is read by a read operation executed by another thread is open to discussion, and requires a deeper understanding of the behavior of the memory system. Arguing about the correctness of a multithreaded code is clearly impossible without a precise answer to this question.

This is a complex issue because it is strongly coupled to—and often in conflict with—the large variety of hardware and software optimizations implemented in current computing architectures to beat what in Chapter 1 was called the *memory wall*. It is not the purpose of this chapter to get involved in a in-depth discussion of this subject, only needed by low-level libraries or compiler writers. Rather, we intend to develop a very qualitative description of what the major issues are—so that readers understand the motivation of the final conclusions of this chapter—trying to conform to A. Einstein's recommendation: *make things as simple as possible, but not simpler*.

The issues to be discussed are of course central for shared memory programming. They have naturally received a lot of attention in the past, and have made some people uneasy towards the implementation of multithreading with libraries on top of thread unaware compilers [19]. Fortunately, barring some pathological cases, applications programmers can safely live with the final conclusions of this chapter.

I strongly recommend readers consult an excellent review paper by S. Adve and H-J. Boenm, *Memory Models: A Case for Rethinking Parallel Languages and Hardware* [20, 21]. Another slightly more advanced but still quite readable discussion can be found in [22].

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00007-1

7.2 CACHE COHERENCY ISSUE

Let us take a detailed look at the cache coherency mechanism. Figure 7.1 shows a SMP platform made of dual-core sockets, with a shared L2 cache memory in each socket.

Let us now imagine that two threads, T1 and T2, running on different sockets have read the A or B data items, and that they both have a corresponding cache line in their L2 caches. Imagine next that one of them, say T1, modifies the value of A. The T1 thread has no problem, because its L2 cache now has the new correct data value. But thread T2 still has the old incorrect value in its L2 cache, so there is a *cache coherency* problem. In order to restore cache coherency, thread T1 that modified A must send a message to all other sockets participating in the process indicating that the cache line that contains A is no longer reliable and must be invalidated. In this way, other threads needing to read A will fetch again the value from the main memory, where hopefully the new updated value is stored.

This cache coherency mechanism requires therefore a persistent communication context among sockets in a SMP platform, and this is the main reason SMP architectures can hardly be extended into the massive parallel domain. It is just too expensive and unrealistic to try to enforce cache coherency among thousands of SMP nodes. Notice also that the mechanism that guarantees cache coherency may have a significant impact on performance that programmers must be aware of. Let us imagine that several threads in a process keep repeatedly updating a shared data item A (a counter, for example). Each time A is updated all cache lines holding A are invalidated, and each subsequent access to A costs the full main memory latency. In this context cache memories are useless, and the performance application may be significantly slowed down.

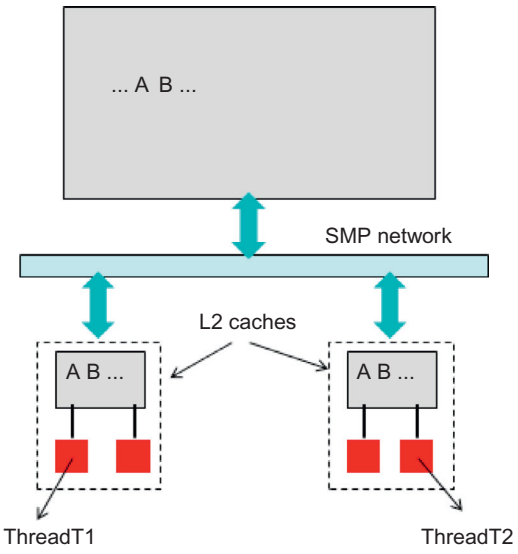

FIGURE 7.1

Illustrating the cache coherency issue.

7.2.1 FALSE SHARING PROBLEM

The cache coherency mechanism may sometimes slow down the application performance even if different threads keep updating different variables. Imagine next—see Figure 7.1—that thread T1 keeps updating A and thread T2 keeps updating B. These are now different variables but, *if they are nearby data items sitting on the same cache line*, everything happens as before, as if shared variable was accessed. Any update of A or B invalidates both A and B. Cache lines are repeatedly invalidated, with the subsequent performance degradation if memory accesses are too frequent. This is the *false sharing* problem, to be discussed in more detail later on.

7.2.2 MEMORY CONSISTENCY ISSUE

The cache coherency mechanism just described provides two benefits to application programmers:

- It guarantees a thread will not read an obsolete value of a data item from its L2 cache, if this data item has been updated by another thread.
- It promises that the new, updated value of the data item will at some point be available in the main memory for a read by another thread.

Cache coherency is therefore a critical part of the memory system, but *it does not settle the memory consistency issue*. It is simply a mechanism that propagates a newly written value in a memory location to its cached copies, by invalidating the cache lines where the copies reside, thereby forcing the associated cores to retrieve the modified value from main memory. There is naturally a time delay in completing a write: the cached copies must be invalidated, and the new value must reside in memory. The cache coherency protocol does not guarantee *when* the write operation completes, and it *does not* answer the following fundamental question:

When a data value updated by one thread is visible in main memory to another threads?

Obviously, a clear answer to this question is needed to be able to discuss the correctness of multithreaded programs. This answer is provided by the *memory consistency model of the programming environment*. This is a rather subtle question because memory consistency is also a hardware issue closely related to the tricks deployed to reduce the negative impact of the large main memory latencies. Different hardware platforms may handle memory consistency in different ways, and compiler writers need to do whatever is needed to adapt the software memory coherency model to the underlying hardware infrastructure.

7.3 WHAT IS A MEMORY CONSISTENCY MODEL?

A memory consistency model of a shared memory multicore architecture provides a formal specification of how the multicore memory system behaves, so that the outcome of a memory read operation is uniquely defined in spite of the asynchronous operation of the different threads. One could imagine enforcing each core to know at every instant what every other core in the system is doing. But this is very expensive in terms of performance and invalidates whatever performance bonus can be expected from the asynchronous execution of several concurrent execution streams. Moreover, most

of the time this information is not needed. Therefore, a memory consistency model is the result of a trade-off between memory consistency guarantees that make programming easier, and performance. A memory consistency model tells programmers what guarantees they can expect from the memory system when accessing shared memory locations. It also provides a few low-level instructions allowing programmers to enforce whatever additional memory co-ordinations are required by the underlying algorithms and not guaranteed by the programming environment. These low-level instructions are called *memory fences*.

7.3.1 SEQUENTIAL CONSISTENCY

The simplest memory consistency model is *sequential consistency*. This model has been formally defined by Lamport [23] as follows:

> A multiprocessor is **sequentially consistent** if the result of any execution is the same if the operations of all the processors were executed one at a time in some sequential order, and the operations of each individual processor appear in this sequence in the same order specified by its program.

In other words: take two threads, and imagine that the operations of each one are represented by a deck of cards, a red and a blue deck. Now push the two decks against each other to interleave the cards. In the final deck, the red as well as the blue cards maintain their initial program order, but the interleaving is totally arbitrary. This arbitrary interleaving is just a manifestation of the asynchronous nature of the threads operation. Nevertheless, there is a precise ordering of the operations of different threads. It is implicitly assumed here that memory writes and reads are instantaneous operations that take immediate effect.

This is a very clear cut memory model. However, serializing the execution of the different instructions streams cannot be the way a parallel code executes in real life. This is why the definition of sequential consistency requires that the result of any execution should come out *as if* the code was executed in the way described above.

In order to see the implications of the sequential consistency requirement, we look at a classic example taken from Chapter 8 in the classic reference on computer architectures [24]. Listing 7.1 shows side by side two code segments executed by threads T1 and T2:

```
int X=0; int Y=0;       // shared variables

Thread T1               Thread T2
– – – – – – – – –       – – – – – – – – –
...                     ...
X = 1;                  Y = 1;
if(Y==0)                if(X==0)
  { ... }                 { ... }
```

LISTING 7.1

Implications of sequential consistency

In this case X and Y are shared variables initialized to 0. Threads T1 and T2 write to X and Y, respectively, and then read the other variable. If, as it is assumed in the sequential consistency statement,

writes always take immediate effect and are immediately seen by other processors, then it is impossible for both if() statements to evaluate to true. Indeed, reaching the statement means that either A or B have been assigned the value 1. If, instead, the write operations are for some reason delayed (and indeed they are, as discussed in the next section), then it is possible that:

- Thread T1 runs ahead of thread T2, so it sets X=1 and reads Y=0.
- Thread T2 is late but not too late, and by the time it reads X the write operation from T1 has not yet completed, in which case it reads X=0.
- Now, both if() statements evaluate to true.

The conclusion is that when the code above is executed, sometimes only one and sometimes both of the two if() statements is executed. Now, one may ask if this behavior should be tolerated, and the answer depends on the programmer's intentions. If the underlying algorithm can leave with both alternatives, then there is no problem. If, instead, the algorithm requires one of the two if() statements to be true, this behavior cannot be tolerated, and sequential consistency must be enforced.

Enforcing sequential consistency leads to the installation of additional constraints between the memory operations in which the same data is being written and read by different threads. The read of T2 must be delayed until the write of T1 is completed. Likewise, the read of T1 must be delayed until the write of T2 is completed. This amounts to establishing "happens before" synchronization operations between the activities of different threads. Chapter 8 will show in detail how "happens before" relations are enforced in the different programming environments.

7.3.2 PROBLEMS WITH SEQUENTIAL CONSISTENCY

There are two major issues to be considered when thinking about sequential consistency.

Memory accesses are not atomic

The simple sequential consistency model above assumes that memory accesses are instantaneous (atomic) and that they complete in the order in which they are issued. This is absolutely not the case. Figure 7.2 sketches the standard hierarchical memory model. At some point, a thread updates a variable sitting in its L2 cache. The cache coherency protocol invalidates the cache lines where the copies reside, thereby forcing the associated cores to retrieve from main memory the modified value. There is naturally a time delay in completing a write: the cached copies must be invalidated, and the new value must reside in memory, in order to be seen by other threads.

Indeed, *writes to main memory can be delayed*. To understand why, remember the expression given in Chapter 1 for the time required to transfer N bytes from processor to memory or vice versa:

$$T = L + \frac{N}{B}$$

where L, the latency, is a characteristic of the memory device, and B, the bandwidth, is a quality factor of the network interconnect. As discussed in Chapter 1, the bandwidth can be improved by enhancing the network quality, but the latency of mass memory chips is of a few hundreds of processor cycles, and we are forced to live with that. It is therefore obvious that one way to improve memory access performance is collecting as much data as possible in a memory block before moving it. In this way, the big latency is paid only once, and not several times. This is the reason why modern processors often

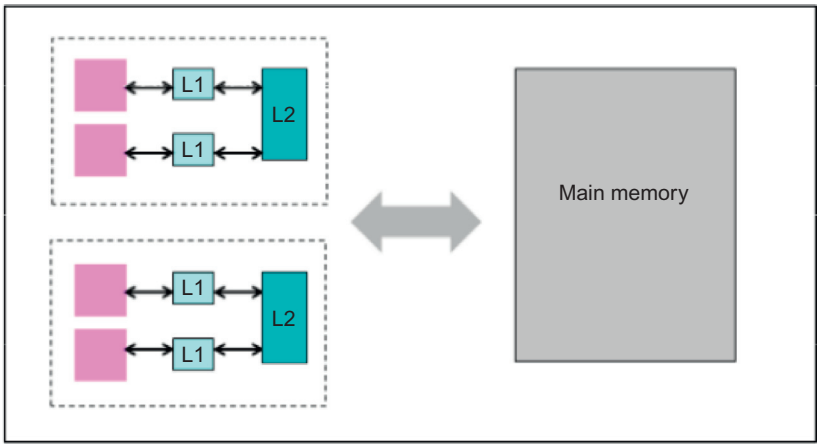

FIGURE 7.2

Hierarchical memory system: read-writes from L2 to main memory are not atomic.

dispose of *write buffers* where data is deposited when, after a write to the L2 cache, new data values must be moved to memory. This introduces a further delay in the completion of the write operation. Likewise, *reads from main memory may be advanced*. The compiler may issue them earlier than needed in order, again, to optimize memory activity. Because of these hardware optimizations, it is not easy to give a precise meaning to *program order* in a multicore system.

Conflicts with sequential compiler optimizations

When the compiler produces the object code in a sequential program, it breaks down the original code into a sequence of low-level machine instructions, and tries to execute them in the most efficient way by exploiting the *instruction level parallelism*. The compiler only guarantees that *the result is consistent with program order execution*. The real code executed is not in program order because of the various optimizations performed by the compiler or the hardware to optimize performance. Many compiler optimizations—such as code motion or loop transformations—or hardware optimizations—like pipelining or multiple issue—lead to overlapping or reorder of memory operations. But for sequential execution streams the compiler can manage to implement the illusion of program order. It is sufficient to respect data and control dependencies among basic instructions. Two memory operations are forced to respect the program order when they are to the same memory location, or when one controls the execution of the other. As long as these data and control dependencies are respected, the compiler and the hardware can freely reorder operations to different memory locations.

In a multicore platform, things become more complex, because:

- The compiler has all the information and the capability of reordering memory accesses and yet guaranteeing sequential consistency *only when dealing with a single, sequential instruction stream*.
- Extending this capability to multiple, simultaneous execution streams is only possible by giving up most of the code optimizations that are at the base of single core performance.

Before discussing the compromises adopted to manage this conflicting context, we take a look at a simple example that shows how the same code, executed in systems with different memory consistency models, can behave in a completely different way. We will, of course, come up later on with the steps that need to be taken to have consistent, portable code.

An amusing example

The example discussed next is the programming of a simple busy wait. The code listed below is supposed to behave as follows:

- There is a shared integer variable, synch, initialized to 0.
- The main thread launches another thread whose only activity is to wait until this shared variable becomes 1. Disregard for the time being the message indicating that it has been released, commented away.
- After launching the waiting thread, the main thread increases synch to 1, and joins the terminating thread. We know that the waiting thread has been released because the join function call returns. If the waiting thread is not released, this function call does not return and the code deadlocks.

The source file Test.C is listed below. We use the SPool utility to manage the waiting thread.

```
int synch;
SPool TS(1);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void ThreadFct(void *P)
   {
   while(synch != 1);
   // printf("\n Thread released\n");
   }

int main(int argc, char **argv)
   {
   synch = 0;

   TS.Dispatch(ThreadFct, NULL);   // launch thread
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   pthread_mutex_lock(&mutex);     // increase synch
   synch ++;
   pthread_mutex_unlock(&mutex);
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   TS.WaitForIdle();               // join
   return 0;
   }
```

LISTING 7.2

Programming a busy wait

Notice that precautions have been taken concerning shared variables: the increment of the integer synch variable is performed with a locked mutex. This may look like overkill, because in this example nobody else is modifying this variable, but we will see later on that this mutex is in any case needed

here for memory consistency reasons. Notice also that *we did not bother to read this variable with a locked mutex*. After all, why should we? A read does not changes the value of synch. We will also see later on that *locking for the read the same mutex used for the write is needed for memory consistency reasons*. This will be the fundamental conclusion of this chapter.

If this code is executed in any Intel x86 architecture (32 or 64 bits), everything is fine and the waiting thread is released. This architecture has a very strict memory consistency model, and the best practices we will discuss in the next sections are not really needed. However, if the code is executed on an IBM Power6 system running AIX the code deadlocks because the waiting thread is not released.

Let me first tell you what I think is going on in the Power6 system. The memory coherency model is more relaxed, and allows some optimizations that would normally not be allowed in more strict models. The compiler, when writing the simple code for the waiting thread, starts by reading synch, whose value is loaded into a register. Since, from the point of view of the waiting thread, nobody else is accessing this variable, the compiler does not bother to read it again from main memory and keeps testing the register value. This is why it will never see the new value written to memory. This clearly shows that a thread can maintain a temporary (or permanent, as in this case) view of memory that is not consistent with the state of the main memory.

How can I guess that this is what is going on? It turns out that, shooting in the dark, I added the print statement initially commented away in the waiting thread function, *and, in doing so, the thread was released*. Now, the print statement has absolutely nothing to do with memory consistency. But an educated guess is that, in order to execute this function call, the compiler needed to use the register holding the *synch* value. Then, when the printf function returns, a new value is read from main memory.

Obviously, programmers cannot live with this erratic, non-portable code with platform-dependent behavior. The basic issues required to ensure memory consistency for robust and portable codes are discussed next.

7.4 WEAK-ORDERING MEMORY MODELS

Strict sequential consistency constrains many hardware and compiler optimizations, and the commonly adopted way out of this conflict is to relax those constraints, while providing at the same time safenets directly or indirectly controlled by the programmer. They are used to restore memory consistency if and when memory operations performed by different threads need to be ordered to guarantee algorithmic integrity.

It is beyond the scope of this book to review the different relaxed memory consistency models implemented by the existing computing platforms. We will only refer to the so-called *weak ordering models*, because they are the ones most commonly adopted in libraries and programming environments. These models classify memory operations into two categories: *data operations*, which are the ordinary reads and writes, and *synchronization operations*, providing the safenets needed to partially restore memory consistency. These models are based on the observation that, typically, memory operations can be reordered within a data region, between synchronization operations, without affecting the correctness of the program. Threads are allowed to maintain a temporary view of memory different from the real state of the memory system. Synchronization operations restore the memory consistency.

7.4.1 MEMORY FENCES

The simplest synchronization memory operation is a *memory fence* instruction. Informally speaking, when such an instruction is issued by a running instruction stream, *no new memory operation is started until this instruction returns, and this happens when all the ongoing memory operations are completed*. At this point, the consistency between the threads view of memory and the real content of the memory system is restored. The result of all the memory operations prior to the memory fence are seen by all threads. It is clear that the presence of a memory fence instruction imposes severe memory-ordering constraints to the compiler, who can no longer reorder preceding (or succeeding) memory accesses beyond (or before) the fence operation point.

In fact, real life is slightly more subtle. Weak-ordering memory models distinguish two types of memory synchronization operations, whose purpose is to reduce as much as possible the constraints they impose, by adapting the operation to the context:

- **Acquire operations**: The memory order constraint is that succeeding operations are not reordered before the fence instruction, but no constraint is imposed on preceding memory operations.
- **Release operations**: The memory order constraint is that preceding operations are not reordered beyond the fence instruction, but no constraint is imposed on succeeding memory operations.

Obviously, the fence operation described first as an acquire-release fence. But it is easy to imagine contexts in which looser memory ordering constraints are useful. Locking a mutex, or reading the value of an atomic variable (discussed in the next chapter), are acquire operations: the thread acquires the right to read a value. Unlocking the mutex, or storing a new value in an atomic variable, are release operations, because they may release a potentially blocked thread. The mutex case is discussed in more detail in the next section. A more precise discussion of these concepts is developed in Chapter 8, when discussing how atomic utilities are used to trigger synchronizations that enforce a precise order between memory operations performed by different threads.

Native libraries or high-level programming environments do not always provide explicit memory fence instructions for programmers. Memory fence instructions are implicit, and they are inserted at the appropriate places in some basic synchronization or management primitives. Then, the end user is instructed about the best practices required to ensure memory consistency. Many programming environments—C++11, Windows, OpenMP, TBB—provide explicit instructions for memory ordering control, in general through their atomic services, discussed in Chapter 8.

7.4.2 CASE OF A MUTEX

The mutex lock-unlock operations are typical places where memory fences are inserted. As shown in Figure 7.3, a memory fence instruction is issued just before the mutex lock function call returns, and just before the mutex unlock function call releases the mutex. Notice that this applies to any kind of mutex. Why are these memory fences needed?

- The fact that the mutex is locked is indicated by a flag inside the mutex object. The first memory fence is needed in order to make absolutely sure that any other thread trying to lock the mutex will see that it is locked. *Mutex locking is an acquired memory operation*. Memory operations testing the mutex state will never be moved up on top of the acquire fence.

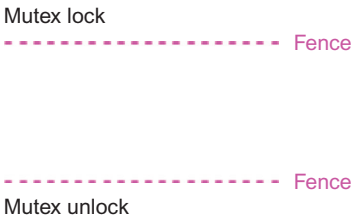

FIGURE 7.3

Memory fences in mutex locking-unlocking operations.

- The second memory fence instruction is also needed to make sure that, when the mutex is unlocked, all the other threads in the system will see the result of all the memory operations performed inside the critical section.
- *Mutex unlocking is an release operation*. Memory operations updating the mutex protected variable will never be moved down below the release fence.

Now, let us try to read the last written value of a shared variable. *This is guaranteed if the read is performed by locking the same mutex used for the write*. The reason for this is that the release fence following the write synchronizes with the acquire fence before the read, establishing a "happens before" relation between the write and the read. This is, admittedly, a somewhat vague explanation. A more precise discussion followed by examples is developed in Chapter 8. Mutex locking, initially introduced to implement mutual exclusion when writing a shared variable, is also needed for reads to guarantee memory consistency.

Mutex locking is not just about mutual exclusion. Mutex locking is also needed to guarantee memory visibility when accessing shared variables.

7.4.3 PROGRAMMING A BUSY WAIT

On the base of this observation, the busy wait code of Listing 7.2 can be re-examined, to make it portable. The main function—which writes to the shared `synch` variable with a locked mutex—is unchanged. The waiting thread function, instead, must read this variable with a locked mutex. Listing 7.3 gives the correct version of this thread function.

```
int synch;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
void *ThreadFct(void *P)
   {
   int my_synch;
   do
      {
```

```
        pthread_mutex_lock(&mutex);
        my_synch = synch;
        pthread_mutex_unlock(&mutex);
        }while(my_synch != 1);
    }
```

LISTING 7.3

Correct thread function for busy wait

A do loop keeps reading memory. We know that the mutex must be locked when the shared variable synch is read. A first idea would be to lock and unlock the mutex outside of the do loop. But this is wrong, because in this case the do loop keeps reading the old value of synch without ever releasing the mutex, and no other thread will ever be able to change its value. The code deadlocks. The correct way of programming the spin wait is given above. The thread function declares a *local* flag (called my_flag in the code). Then, the waiting thread enters a do loop in which:

- The mutex is locked-unlocked just to copy the predicate value to the local flag my_flag.
- Then, the value of my_flag is tested to decide if another iteration is required.
- Since the mutex is unlocked at each iteration, other threads have a chance of acquiring it.

Example: Waiting for the IO task using a spin wait

As an example of the utilization of spin waits, the previous IoTask.C example in Chapter 6 is re-examined. The main thread waits for the completion of a long IO operation delegated to a worker thread. OpenMP threads are used in this example.

```
bool     flag;               // predicate
omp_lock_t mylock;           // mutex (guards predicate)

// Code for workers tasks
// - - - - - - - - - - - - - - - - - - - - - -
void io_task()
   {
   Timer T;
   T.Wait(2000);  // perform IO operation
   std::cout << "\n IO operation done. Signaling" << std::endl;

   // change flag value
   omp_set_lock(&mylock);
   flag = false;
   omp_unset_lock(&mylock);
   }

void main_task()
   {
   bool my_flag;
   do
     {
```

Continued

```
      omp_set_lock(&mylock);
      my_flag = flag;
      omp_unset_lock(&mylock);
      }while(my_flag==true);
   std::cout << "\n Main task released" << std::endl;
   }

void TaskFct()
   {
   int rank = omp_get_thread_num();
   if(rank==0) main_task();
   else io_task();
   }

int main(int argc, char **argv)
   {
   flag = true;
   omp_set_num_threads(2);
   #pragma omp parallel
      { TaskFct(); }
   return 0;
   }
```

LISTING 7.4

IoTaskOmp.C

In this code:

- main() initializes the predicate to true, sets the number of threads to 2, and launches an OpenMP parallel section where all tasks execute the function TaskFct().
- TaskFct() gets the rank of the executing thread, and calls main_task() if rank==0, or io_task() if rank==1.
- The IO task is identical to the one discussed before: when the Timer returns after 2 s, this task toggles the value of the predicate to false.
- The main task executes a spin wait as long as the predicate is true, and prints a message when the thread is released.

7.5 PTHREADS MEMORY CONSISTENCY

Here are the memory consistency engagements of the Pthreads library, which result from the existence of implicit memory fences in different library functions:

- Thread creation: memory values a thread can see when it calls pthreads_create() can also be seen by the new thread. Any data written to memory after the call may not be seen by the new thread.

- Mutex unlock: whatever memory values a thread can see when it unlocks a mutex—directly, or waiting on a condition variable—can also be seen by another thread that later locks the same mutex.
- Condition signal or broadcast: whatever memory values a thread can see when it signals or broadcasts a condition variable, can also be seen by a thread that is awakened by that signal or broadcast.
- Thread join: whatever memory values a thread can see when it terminates can also be seen by the thread that joins with it by calling pthread_join().

Memory consistency safenets are therefore installed whenever threads are created or joined, and at the two basic synchronization primitives. Looking back at the simple fork-join examples presented in previous chapters, you can verify that these memory consistency rules have been implicitly applied in those cases.

7.6 OpenMP MEMORY CONSISTENCY

OpenMP puts implicit memory fences in the following synchronization contexts:

- At a barrier synchronization point
- On entry and exit of parallel and critical regions
- At omp_set_lock and omp_unset_lock regions
- At omp_set_nest_lock and omp_unset_nest_lock regions
- At omp_test_lock and omp_test_nest_lock regions, if the call succeeds and the region causes the lock to be set or unset
- Immediately before and after every task scheduling point

Again, the basic idea is the same: memory consistency safenets installed in fork-join sections, and in the basic synchronization primitives used by Open MP: mutex locks and barriers. We will see in Chapter 10 what a task synchronization point is.

7.6.1 FLUSH DIRECTIVE

OpenMP has, in addition, a flush directive that plays the role of an *explicit* memory fence. There general form of the directive is:

```
#pragma omp flush(a, b, ...)
```

LISTING 7.5

Correct thread function for busy wait

and, in this case, memory consistency is restored for variables (a, b ...). There is also a no argument form that acts exactly like a memory fence, restoring memory consistency for all shared variables in memory. Using this explicit memory fence instruction is delicate, and the official OpenMP documentation takes a lot of precautions in presenting its potential usefulness. One of the subtle points is that the flush directive itself can be reordered by the compiler, and therefore this directive cannot

be manipulated with shaky hands. Users are encouraged to rely as much as possible on the implicit memory fences already incorporated in the synchronization primitives.

The flush directive has clearly been incorporated in OpenMP to facilitate some low-level optimizations, but, as with the atomic classes discussed in Chapter 8, it falls in the area of advanced low-level optimization tools whose usage is often useful, but never mandatory. Any memory consistency context can always be correctly implemented with mutex locking, as discussed before. It is nevertheless useful to dispose of a qualitative idea of the way other slightly more refined approaches operate, based on atomic utilities. This is the subject of next chapter. The memory consistency mechanisms discussed before will be clarified by discussing the `std::atomic<T>` class proposed by the C++11 thread standard.