CHAPTER

ATOMIC TYPES AND OPERATIONS

8

8.1 INTRODUCTION

Atomic operations were introduced in Chapter 5 as a useful and fast alternative for implementing mutual exclusion among basic shared variables. They are guaranteed to be performed in an indivisible way, as if they were a unique machine instruction. This is, in fact, the way they are used in OpenMP. In other programming environments—Windows, TBB, C++11—atomic operations have a broader role: they enable programmers to enforce the ordering of memory operations of different threads and to construct in this way custom synchronization patterns. This is the subject of this chapter. The memory consistency concepts discussed in the previous chapter are essential for the discussion that follows.

The TBB and C++11 libraries provide C++ classes defining *atomic data types*, on which a set of atomic operations are defined. Windows does not have explicit atomic data types, but a number of C library functions provide an equivalent service. These atomic utilities are powerful weapons in skilled hands, opening the way to the design of efficient custom synchronization utilities enhancing multithreading performance.

As stated in Chapter 5 any synchronization context can always be managed with the two basic synchronization primitives: mutual exclusion and event synchronization. Nevertheless, a qualitative understanding of the way atomic utilities operate is useful knowledge. It is possible in this way to understand the basic ideas behind the so-called lock-free algorithms, whose purpose is to control race conditions in shared data accesses without blocking the participating threads. Later chapters explore also the possibility of substantially enhancing the performance of some realistic applications with the help of simple utilities based on atomic variables.

This chapter covers the basic atomic concepts and ideas, together with a few examples. The atomic classes proposed by the C++11 thread library and by TBB are discussed first. Then, the Windows atomic utilities are discussed. A very complete and exhaustive discussion of the C++11 atomic standard can be found in A. Williams book, *C++ concurrency in action* [14].

8.2 C++11 STD::ATOMIC<T> CLASS

The C++11 standard defines the `std::atomic<T>` class. The template argument `T` is a generic type: bool, integer, or pointer type, another basic type—like a double, for example—or a user-defined type fulfilling certain criteria. Explicit atomic operations can then be programmed in a simple way. However, not all the operations defined by the class member functions can be applied to all types: there are indeed

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00008-3

a few specializations for specific types. In what concerns user-defined types, they must not have any virtual function or virtual base classes, and they have to be simple enough so as to use the default, compiler-generated copy and assignment operators.

8.2.1 MEMBER FUNCTIONS

In the discussion that follows, X and Y are type T atomic data items, and x, y, are type T ordinary data items. The member functions discussed below show an optional last argument M that corresponds to a *memory-ordering option*, related to memory consistency requirements that will be specified in detail later on. A default value is adopted if this optional argument is omitted.

STD::ATOMIC<T> INTERFACES

std::atomic<T> X

- Constructor.
- Creates an atomic variable X of type T.

bool X.is_lock_free()

- Returns true if the internal implementation of the atomic variable of type T is not emulated by mutex locking.
- Returns false if the atomic variable of type T is emulated by using mutex locking.

x = X.load(M)

- Reads in x the internal value of X.

void X.store(x, M)

- Initializes the internal value of X to x.

x = X.exchange(y, M)

- Sets the internal value of X to y.
- Returns the old value of X in x.

bool X.compare_exchange_strong(T& e, T x, M)

- Atomically compares the expected value e to the value stored in X, and stores x in X if equal.
- Returns true if exchange succeeds, false otherwise.

x = X.fetch_add(y, M)

- Atomically adds y to the value stored in X.
- Returns the old value stored in X.
- Alternative notation: X+= y;

x = X.fetch_sub(y, M)

- Atomically subtracts y to the value stored in X.
- Returns the old value stored in X.
- Alternative notation: X -= y;

x = X.fetch_or(y, M)

- Atomically ors the value stored in X with y.
- Returns the old value stored in X.
- Alternative notation: X |= y;

x = X.fetch_and(y, M)

- Atomically ands the value stored in X with y.
- Returns the old value stored in X.
- Alternative notation: X &= y;

x = X.fetch_xor(y, M)

- Atomically xors the value stored in X with y.
- Returns the old value stored in X.
- Alternative notation: X ≙y;

X++; X−−;

- Atomically increments or decrements the value stored in X.

The compare_exchange_strong() member function quoted above has another version with the same signature, called compare_exchange_weak. Notice that many "read and modify" atomic operations return the old value of X. The reason is that when these basic operations are used to build higher level synchronization constructs, the initial value of X is sometimes needed, as some of the examples that follow later on in the chapter will show (Table 8.1).

A few more comments concerning the properties of this class:

- The atomic<T> class, like the basic data types, has no explicit constructor. The only way to create an atomic variable is to declare it in the way indicated above. The C++ initialization rules for classes without constructors apply here. The atomic data item is initialized to 0 (if integer) or NULL (if pointer) when declared as a global variable with file scope, as static data item inside a function, or as a data member in a class.
- When pointers are used, the arithmetic operations comply with the *pointer arithmetic* specifications. The ++ operation, for example, increments an atomic T* of sizeof(T).
- Likewise, when an integer n is added to an atomic pointer T*, the pointer value is incremented by n.sizeof(T)

Example 1: Monte-Carlo computation of π

As a first simple example, let us go back to the Monte-Carlo computation of π discussed in previous chapters. Remember that two threads were counting the number of accepted events, and that the main issue was to communicate the partial acceptances to the main thread. The simplest thing to do now is

Table 8.1 Available Operations for the Different Atomic Types

	Atomic Operations			
Operation	**bool**	**integer**	**pointer**	**basic or user**
is_lock_free	X	X	X	X
load	X	X	X	X
store	X	X	X	X
exchange	X	X	X	X
comp_exch	X	X	X	X
fetch_add, +=		X	X	
fetch_sub, -=		X	X	
fetch_or, |=			X	
fetch_and, &=			X	
fetch_xor, ≙			X	
++, −−		X	X	

to define a atomic<long> global counter, incremented by each thread each time an accepted event is registered. Then, the main thread just uses the final value of this counter. The new version of this code is in the source file McAtomic_S.C. Listing 8.1 displays the new thread function.

```
#include <SPool.h>
#include <atomic>
SPool TH(2);              // set of two threads
long nsamples;            // number of MonteCarlo events per thread
std::atomic<long> C;      // atomic<long> to accumulate acceptances

void *thread_fct(void *P)
   {
   double x, y;
   int rank;

   rank = TH.GetRank();
   Rand R(rank*999);
   for(int n=0; n<nsamples; n++)
      {
      x = R.draw();
      y = R.draw();
      if((x*x+y*y) <= 1.0 ) ++C;  // accumulate in atomic C
      }
   }
```

LISTING 8.1

McAtomic_S.C (partial listing)

After joining the threads, main() uses the C value to compute π. A substantial amount of contention has deliberately been introduced in this example: the atomic counter is incremented a huge number of times. We know that a much better strategy would be to have each thread increment a private local counter, and then accumulate only once each final private count in the atomic counter. Nevertheless, the performance of this program is quite acceptable when compared to the CalcPi.C performance, where partial results are accumulated in local variables.

The excessive contention on the atomic integer is much less harmful than the excessive contention in mutex locking. The reason is that atomic operations are implemented using very low-level hardware and operating system facilities specially designed to implement thread synchronizations. This can be easily verified in this example: if an arbitrary additional input is added to the command line, the program accumulates acceptances in a mutex protected long instead. The atomic version runs several times faster than the locked mutex version.

Example 1: McAtomic_S.C

To compile, run make atest. The number of threads is hardwired to 2. Adding any arbitrary input to the command line, mutex locking replaces the atomic operation.

8.3 LOCK-FREE ALGORITHMS

Atomic classes make can be used to perform a thread-safe update of a shared data set without introducing critical sections, by implementing lock free algorithms, where threads may possibly perform redundant work, *but they are never blocked, waiting for a mutex availability*. Performing additional redundant work may in many cases be a better trade-off than the serialization enforced by the critical sections. The difference in both approaches is the following:

- Mutual exclusion is a *pessimistic* approach, planning for the worst. In acting on the data set, the programmer strategy is: *a race condition could possibly happen here, so we make sure that it never happens*.
- Lock-free algorithms implement an *optimistic* approach, planning for the best. In acting on the data set, the programmer strategy is *we go ahead, and if a race occurs we start all over again*. Of course, the whole point is having the right tool to detect that a race condition has indeed occurred.

The lock-free logic is discussed next. This is not really application programming, but the very instructive programming idiom that follows will certainly improve your understanding of multithreaded programming. This discussion demonstrates the power of the atomic<T> classes.

8.3.1 UPDATING AN ARBITRARY, GLOBAL DATA ITEM

Consider the update of a basic or user-defined type T. The fundamental tool for detecting race conditions is the compare_and_exchange member function, usually called CAS, for *Compare and Swap*. This is the basic tool enabling the implementation of lock-free algorithms. In most systems it is directly supported by hardware. The x86 and Intel 64 instruction sets have a unique CAS instruction that works naturally for integers and pointers. In PowerPC processors, the CAS operation is implemented by two instructions. The atomic<T> compare_and_exchange member functions clearly rely on these low-level hardware primitives, at least for integers and pointers. In C++11, the CAS operation is extended to arbitrary types, but this is not the case for Windows and TBB, where it remains restricted to integer or pointer types.

Figure 8.1 shows the lock-free algorithm for the update. Let old_value be the initial value of a data item, allocated in the heap and referenced by an atomic pointer P.

- In step 1, the initial content of P is copied to a local variable x.
- In step 2, another local variable y is allocated in the heap, referenced by an ordinary pointer Q. This variable is initialized to the new value that needs to be loaded in P.
- In step 3, a complex operation involving a comparison and an exchange is performed atomically: P is compared to its initial value, and if there has been no changes, P and Q are exchanged.
- At the end of this process, P references the updated new value.

In this algorithm, it is the CAS operation that detects possible race conditions. A race condition occurs if, in the lapse of time in which the active thread saves the initial value and allocates the new value, another thread—concurrently updating P—is faster and completes its own update. In this case, P has been modified, the comparison fails, and the update does not take place. Indeed, a race condition has occurred, and knowing that the race is lost the update is aborted and eventually started again from

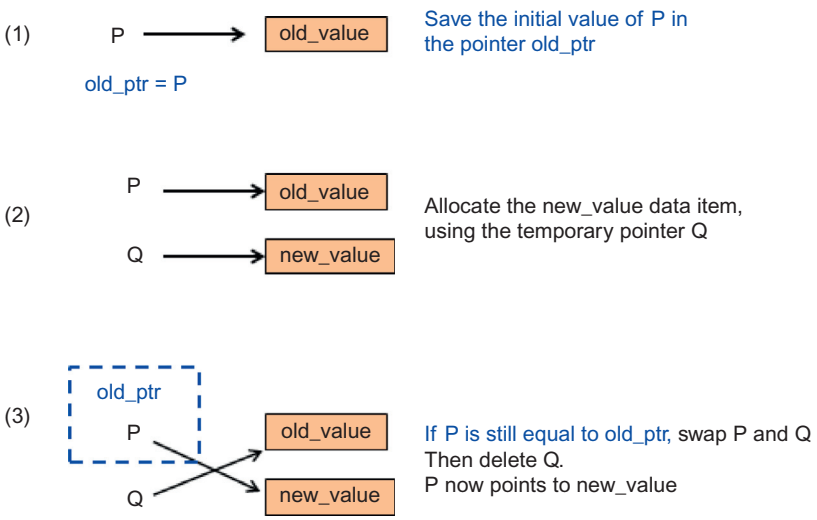

FIGURE 8.1

Lock-free algorithm for updating an arbitrary data item.

scratch. If the comparison succeeds and the exchange takes place, the atomic nature of this combined operation prevents it from being corrupted by another competing thread.

8.3.2 AREDUCTION CLASS

The AReduction class—for *Atomic Reduction*—has exactly the same public interfaces as the Reduction class introduced in Chapter 5, with a different implementation: a lock-free algorithm instead of mutex locking. This new class encapsulates an *atomic pointer* T* to an arbitrary data type, allocated in the heap.

```
#include <atomic>
template<typename T>
class AReduction
   {
   private:
    std::atomic<T*> P;

   public:
    AReduction() { P = new T(); }

    ~AReduction() { delete P; }

    T Data() { return *P; }

    void Reset()
       {
```

```
        T value  = *P;
        *P -= value;
        }

     void Update(T d)
        {
        T *oldptr, *newptr;
        do
          {
          oldptr = P;
          newptr = new T(*P+d);
          }while(P.compare_exchange_strong(oldptr, newptr)==false);
        }
    };
```

LISTING 8.2

AReduction class—Atomic, lock-free accumulator

Notice the simplicity of the Update(T d) member function: it executes a do loop that keeps iterating as long as the CAS operation fails because a race condition is detected. Notice also how the return value of compare_exchange() plays a critical role in the algorithm. The member function Data() returns the value of the data referenced by P, and the Reset() member function reinitializes the internal data to 0, so that a new reduction operation can be started.

This class operates exactly like Reduction.h. It accumulates values, and returns the result of the reduction via the Data() function call. Pointer manipulations are hidden to the user. The source code can be consulted in the file AReduction.h. The class is tested in ScaProd_S.C.

```
SPool TH(2);                  // set of two threads
AReduction<double> D;         // lock free double accumulator
double A[VECSIZE];
double B[VECSIZE];

void *thread_fct(void *P)
   {
   double d;
   int beg, end;
   beg = 0;                     // initialize [beg, end) to global range
   end = VECSIZE;
   TH.ThreadRange(beg, end);    // [beg, end) is now range for this thread
   for(int n=beg; n<end; n++)
      {
      d = A[n]*B[n];
      D.Update(d);
      }
   }
```

LISTING 8.3

Partial listing of ScaProd_S.C

Example 2: ScaProd_S.C

To compile, run make scaprod_s. The number of threads is hardwired to 2.

In running the example, you will observe that this code exactly reproduces the results of all the previous thread-safe versions of this calculation. Once again, unnecessary contention is deliberately introduced, but this code has better performance than the previous, mutex-based inefficient version, DotProd1.C.

8.4 SYNCHRONIZING THREAD OPERATIONS

Our interest moves next to the way atomic operations are used to implement synchronization patterns, by enforcing some degree of ordering among memory operations performed by different threads. The discussion that follows focuses on the C++11 library, but all the ideas and concepts developed here apply also, with minor modifications, to the TBB atomic class and to the Windows atomic utilities.

Let us imagine that a thread T1, updating some data structure, needs to enforce the fact that another thread T2 only accesses the same data structure after the update is completed. In other words, a *happens before* relationship must be established between the activities of threads T1 and T2.

This "happens before" relationship can be implemented as follows. A shared atomic<bool> X is introduced, with an initial false value. When T1 finishes its update, it stores the value true. T2 keeps loading the atomic flag until the value true is retrieved. At this point, T2 accesses the data structure. This is shown in Listing 8.4.

```
std::atomic<bool> X;

void ThreadFct1()         // thread T1
   {
   ...
   Operation A;
   X.store(true);
   ...
   }

void ThreadFct2()         // thread T2
   {
   ...
   while(X.load()==false);
   Operation B;
   ...
```

LISTING 8.4

"Happens before" relation between A and B

The reason why this code works is that *the store operation in thread T1 synchronizes with the load operation in thread T2 that reads the same stored value in the atomic variable*. The synchronization therefore occurs when T2 reads true. One can imagine that at this point a memory fence among the two

threads is established, preventing operations in T1 before the store to be moved beyond the fence, and operations in T2 after the load to be moved before it. This mechanism establishes a "happens before" relation between operations A and B in the two threads.

Inside a given thread, there is a natural "happens before" relation among successive operations, provided by program order. Moreover, "happens before" relations are transitive. Therefore, the code above implements a "happens before" relation between any operation before the store in T1 and any other operation after the load in T2.

Atomic operations induce in this way synchronization relations that enforce some degree of memory consistency, with a refined control provided by the optional second argument `M` in the member functions, called a *memory order option*. In the example above, the default value of this option—which is `sequential consistency`—is used; it constitutes the strongest possible enforcement of memory consistency. Other explicit options can be used to reduce this level of enforcement, with the purpose of avoiding the cost of additional synchronizations that are not strictly needed in a particular context. The discussion that follows clarifies these issues.

8.4.1 MEMORY MODELS AND MEMORY ORDERING OPTIONS

Three memory models proposed by C++11, which, together with the memory ordering options that can be used in each one of them, are described next. Memory order options are symbolic constants passed as second argument to the atomic class member functions. As stated before, the same memory models are implemented in TBB and Windows, with minor implementation differences.

- **Sequential consistency**: This memory model imposes the strongest memory order constraints. In fact, there is more to sequential consistency than the happens before relation exhibited in Listing 8.2. Broader global synchronizations are implemented, forcing all threads to see all the sequential consistent synchronizations in the program happening in the same order. There is, in a given program, a unique global order of sequentially consistent atomic operations, seen by all the threads in the process.
  - `memory_order_seq_cst`: This ordering enforces sequential consistency, preventing preceding (or succeeding) memory operations to be reordered beyond (or before) this point. But there are also subsidiary global synchronizations as discussed above. An example will be given next. *This is the default value for the second argument in member functions.*
- **Acquire-release**: In this model, stores and loads still synchronize as described above, but there is no global order. Different threads may see different synchronizations happening in different orders. Memory order options are:
  - `memory_order_release` Prevents preceding memory operations from being reordered past this point.
  - `memory_order_acquire` Prevents succeeding memory operations from being reordered before this point.
  - `memory_order_consume` A weaker form of the acquire option: memory order constraints only apply to succeeding operations that are computationally dependent on the value retrieved (loaded) by the atomic variable.

Table 8.2 Memory Order Options for Atomic Operations

Options for Atomic Operations	
Operation	**Memory Options**
Store	memory_order_seq_cst memory_order_release memory_order_relaxed
Load	memory_order_seq_cst memory_order_acquire memory_order_consume memory_order_relaxed
Read-modify-write	all available options

  - memory_order_acq_rel. Combines both release and acquire memory order constraints: a full memory fence is in operation.
- **Relaxed**: In this model, the basic store-load synchronization described above does not occur, and no "happens before" relations across threads are established.
  - memory_order_relaxed. There is no ordering constraint. Following operations may be reordered before, and preceding operations may be reordered after the atomic operation.

Finally, it is important to observe that not all memory ordering options are adapted to all member functions. Atomic operations are of three kinds: *load, store*—the load() and store() member functions—and *read_modify_write*—all the other member functions. Table 8.2 lists the memory ordering options available to each type of atomic operation.

The rest of this section provides a very qualitative discussion of the differences between the different memory ordering options, so that readers can feel more or less at ease when dealing with references on this subject. A couple of examples will also be examined. There are, in this subject, a substantial number of subtleties that are beyond the scope of this book. Readers willing to go deeper into this subject should consult the exhaustive discussion presented in [14].

8.4.2 USING THE RELAXED MODEL

Relaxed memory ordering options can be used when the atomic operation is not involved in a synchronization activity, and all that is expected from them is mutual exclusion. In the first example—Listing 8.1—where an atomic counter is incremented in the Monte Carlo computation of π, different threads increment the counter, but the order in which this happens is totally irrelevant.

The relaxed model is not easy to use in complex situations. The C++ atomic services also provide programmers with the ability to introduce explicit memory fences, and a relaxed model with explicit fences may help library writers perform refined optimizations. Explicit C++11 fence interfaces will not be discussed any further. Again, a very exhaustive discussion of all these issues is developed in [14].

FIGURE 8.2

Global order among sequentially consistent operations.

8.4.3 GLOBAL ORDER IN SEQUENTIAL CONSISTENCY

All sequentially consistent atomic operations are executed in a global order that is seen by all the threads in the process. Figure 8.2—inspired by an example proposed in the Boost documentation—shows two threads executing sequentially consistent operations. They do not need to be using the same atomic variable; different variables will do as well. When the program is executed, an absolute order is established, which means that *either A happens before D—the order in the figure—or C happens before B*.

Sequential consistency establishes therefore "happens before" relations between remote operations—with eventual performance penalties—that may or may not be needed in each particular application to preserve the algorithm integrity.

8.4.4 COMPARING SEQUENTIAL CONSISTENCY AND ACQUIRE-RELEASE

The same synchronization relation exposed in Listing 8.3, leading to a "happens before" relation between operations A and B, can be implemented by enforcing release memory order in the store and acquire memory order in the load, as shown in Listing 8.5. If all the program requires is the establishment of this local "happens before" relation between A and B, then the acquire-release model is largely sufficient to preserve the algorithm integrity.

```
#include <atomic>

std::atomic<bool> B;

void ThreadFct1()          // thread T1
   {
   ...
   Operation A;
   B.store(true, memory_order_release);
   ...
   }
```

Continued

```
void ThreadFct2()          // thread T2
   {
     ..
   while(B.load()==false, memory_order_acquire);
   Operation B;
   ...
```

LISTING 8.5

"Happens before" with acquire-release

The only way to see the difference with sequential consistency is to examine the additional side effects when more than one synchronization is programmed. Figure 8.3—inspired from an example in [14]—shows two independent load-store synchronizations: threads T1-T2 establish a "happen before" relation between A and B, and threads T3-T4 establish an independent "happens before" relation between C and D.

If the acquire-release model is used, this is the end of the story: there is no global order established among the two synchronization operations. If, instead, the sequential consistency model is used, all four threads must see the two synchronization operations happening in the same order, and this leads to additional constraints on the way the threads operate.

In order to see this point, threads T2 and T4 are asked, after the synchronizations take place, to load the values of Y and X, respectively. Let us assume that T2 reads the value false. At this point, T2 knows that the T3-T4 synchronization comes after the T1-T2 one, and that the store in X happened before the store in Y. But this order must be seen by all threads, including T4, which means that, when T4 loads X, *it must necessarily read the value true*, because the store in X happened before D. To sum up:

- In the sequential consistency model, threads T2 and T4 cannot both read the value false.
- In the acquire-release model, the two synchronizations are totally independent, and threads T2 and T4 can read any value.

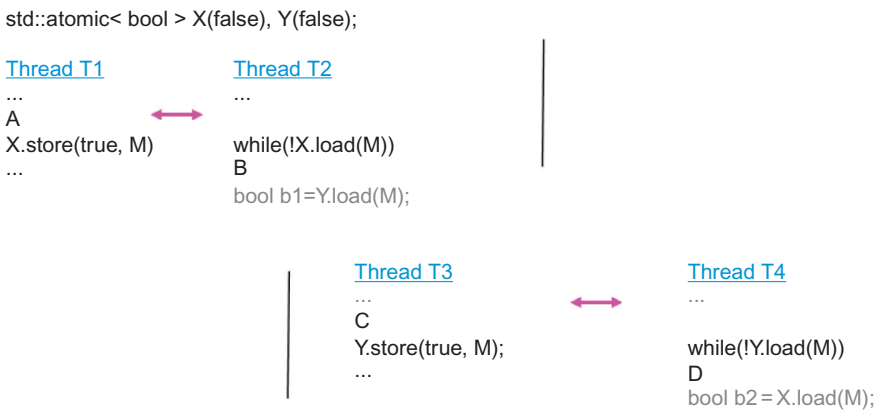

FIGURE 8.3

Two independent "happens before" relations.

8.4.5 CONSUME INSTEAD OF ACQUIRE SEMANTICS

The consume option is an optimized version of the acquire. When an atomic load operation takes place:

- If acquire semantics is used, no memory operation following the load can be moved before it.
- If consume semantics is used, the previous memory order constraint only applies to operations following the load *that have computational dependence on the value read by the load*. This more relaxed requirement reduces the constraints imposed on the compiler, who can now reorder memory operations he could not reorder before, which may result in enhanced performance.

The consume semantics is typically used with atomic pointers. Listing 8.6 shows an example in which a thread transfers to another thread the value of a shared double.

```
#include <atomic>

std::atomic<double*> P;    // initialized to NULL
double               d;

void ThreadFct1()         // thread T1
   {
   ...
   d = 5.25;
   P.store(&d, memory_order_release);  // store pointer to d
   ...
   }

void ThreadFct2()         // thread T2
   {
     ..
   double *p;
   do
      {
      p = P.load(memory_order_consume);
      }while(p==NULL);
   // other operations
   double x = *p;       // use the loaded pointer
   ...
   }
```

LISTING 8.6

"Happens before" with consume-release

In this case, the only "happens before" relation that holds is the one between the initialization of d in thread 1, and the retrieval of is value by the load operation on P in thread 2.

8.4.6 USEFULNESS OF MEMORY_ORDER_ACQ_REL

The memory_order_acq_rel option can be useful to simplify transitive "happen before" relations. Let us consider the following scenario involving three threads:

- Thread T1 executes an operation A followed by a release operation on an atomic variable S1.
- Thread T2 executes a acquire operation on S1, followed later on by another release operation an atomic variable S2.
- Finally, thread T3 executes an acquire operation on S2, followed by an operation B.

Then, operation A in T1 "happens before" operation B in T3. This synchronization pattern can be simplified, using the acquire-release memory order option. First, only one atomic variable S is needed. Then,

- Thread T1 executes an operation A followed by a release operation on S.
- Thread T2 executes any read_and_modify operation on S, with acq_rel semantics.
- Finally, thread T3 executes an acquire operation on S, followed by an operation B.

8.5 EXAMPLES OF ATOMIC SYNCHRONIZATIONS

Some examples of high-level synchronization utilities are proposed next, to be used in later chapters. They illustrate the way atomic operations implement custom synchronization patterns. These examples are proposed for pedagogical purposes, in order to add some real content to the previous discussion.

8.5.1 SPINLOCK CLASS

When discussing mutual exclusion, we observed that Pthreads proposed a special kind of mutex, called spinlock, in which threads keep executing a busy wait in user space when waiting to lock the mutex, instead of moving away to a blocked state waiting to be waken up.

It was also observed that there is no native spin lock in C++11, but it is now very easy to build one. Here is a SpinLock class taken from an example proposed in the Boost documentation, providing the traditional Lock() and Unlock() member functions.

```
class SpinLock
   {
   private:
    std::atomic<bool> _state;

   public:
    SpinLock() :
       {
       _state.load(false, memory_order_acquire);
       }

    void Lock()
       {
       while(_state.exchange(true, memory_order_acquire) == true)
       }

    void Unlock()
       {
       _state.store(false, memory_order_release);
```

```
        }
    };
```

LISTING 8.7

SpinLock class

Let us examine in detail the way this streamlined code operates:

- The internal state of a shared SpinLock object used to enforce mutual exclusion is an atomic bool having two values: true when locked and false when unlocked.
- The Lock() member function calls bool exchange(true, M), which changes the internal state of the atomic variable to true and returns the old value.
- This all happens inside a while loop, which in fact executes a busy wait: it keeps replacing the internal state by the true value, and this loop keeps executing as long as there is no effective replacement because the old value was already true. The while loop breaks when it finds a false state because another thread has released the mutex.
- The exchange operation adopts an acquire semantics, needed to catch a change of state previously executed by another thread.
- The Unlock() member function just stores the false value in the atomic variable state. It adopts the release memory order, so that any atomic variable access that comes after can see the new state value.

This is a very simple and elegant implementation of a spin lock. It correctly enforces mutual exclusion *as long as all threads respect the lock-unlock protocol*. In the code above, there is no notion of lock ownership: nothing prevents a thread from unlocking a spin-lock it has not previously locked, in which case mutual exclusion is corrupted. The spin-lock class in the vath library prevents this from happening by incorporating thread identities in the internal state (see the SpinLock.h source file). This class is used in Chapter 9 to implement a spin barrier. Further examples are also found in this chapter.

This class is fully portable. It is implemented in C++11 and Windows using the atomic services, as explained above. In Pthreads, lacking atomic services, this class is just a wrapper for the native spinlock_t mutex discussed in Chapter 5.

Example: TSpinLock.C

An example follows in which this class is tested in a context of very high mutual exclusion contention: the database search emulation example discussed at the end of Chapter 3. Worker threads keep calling a random number generator until they get a value close to a target value. When this happens, a global flag is toggled to inform the other threads that the result is available, and all infinite loops stop. If all the threads keep checking the global flag at each iteration, performance is extremely poor. The TSpinLock.C alleviates this performance issue while keeping a high amount of mutual exclusion contention by checking the global flag only every 10 iterations. This raises, of course, another problem, because now it is possible for two or more threads to find simultaneously a value close to the target. It is a good exercise to understand how this issue is resolved: another mutex is introduced to catch the first value available. The source file explains in detail the programming strategy.

Example 3: TSpinLock.C

The database search case. Two worker threads are used. To compile, run make tsplock.

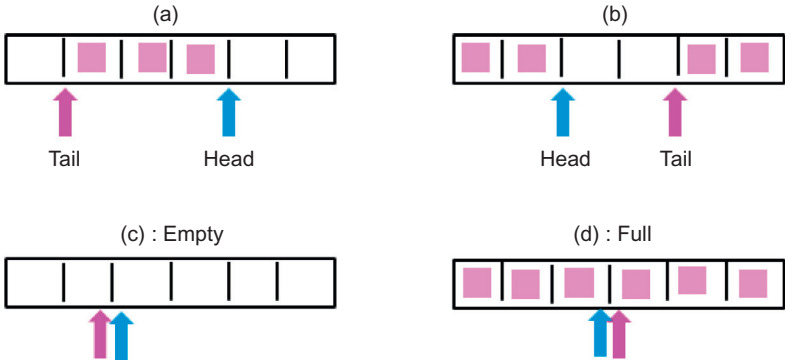

FIGURE 8.4

Circular buffer.

When the code is executed with the configuration given in the example, the Pthreads version take about 3 s, while the versions using the atomic implementations take about 2 s.

8.5.2 CIRCULAR BUFFER CLASS

The next example is a simple, but rather subtle, exercise in atomic synchronizations: Readers are strongly encouraged to take a look at it.

A circular buffer is a utility used to transfer successive data values from a producer thread to a consumer thread, who retrieves the data in FIFO (first in first out) order. This kind of data structure will be used when pipelining threads, a subject discussed in detail in Chapter 15. A producer thread performs some partial operation on a data set (think of one row of a matrix) and stores it in the buffer as a data identifier (think of the row index) to inform the consumer thread that it can go ahead and complete its part of the operation. The circular buffer discussed next assumes that there is one producer and one consumer thread. This is sufficient to implement the pipeline concurrency pattern.

The circular buffer, implemented by a template `RingBuffer` class, is shown in Figure 8.4. A `RingBuffer` object stores an array of type `T` data items, `T` being the template parameter of the class. The buffer size `Size` is also passed as a template parameter. Another private data members are `head` and `tail`, the array indices where the next insertion—or extraction—take place. The circular nature of the array means that `head` and `tail` grow modulo `Size`: the next value to `Size-1` is 0. Figure 8.4 shows some typical configurations: (a) and (b) show a partially filled buffer, (c) shows an empty buffer, and (d) shows a filled buffer.

Circular buffers are useful for two reasons. First of all, this is the only practical way of handling a context in which the buffer size cannot be guessed "a priori": there may be an enormous number of insertions and extractions. The second, most important, reason is that the circular buffer helps to regulate the workflow between the producer and the consumer activities:

- The producer thread calls, to insert a new value, a `Push()` function that operates as follows:
  - Reads the current value of `head`.
  - If the buffer is full, returns `false`.

  - Stores the input data in the buffer at the head position, stores in head its next value, and returns true.
- The consumer thread calls, to extract a new value, a Pop() function that operates as follows:
  - Reads the current value of tail.
  - If the buffer is empty, returns false.
  - Reads the requested data in the buffer at the tail position, stores in tail its next value, and returns true.

The finite buffer size prevents therefore the producer thread from running too much ahead of the consumer thread: insertions are refused when the buffer is full because the consumer is late in extracting data. It is understood that Push() and Pop() need to check if the buffer is full or empty; Figure 8.4 shows that in both cases head==tail. In order to distinguish both configurations, another atomic integer C is introduced, to be increased—or decreased—whenever head—or tail—start a new round. When the buffer is empty, head and tail have performed the same number of rounds, and C=0. When the buffer is full, head is one round ahead, and C=1. The counter C is atomic only for mutual exclusion reasons: both threads are modifying it.

Need for synchronization

The next point is to understand when and how memory operations must be synchronized. In ordinary configurations, with a partially filled buffer, synchronizations are hardly needed: Push() and Pop() operations are acting on *different* array elements. Problems arise when the buffer is empty or full, because in this case both operations may end up acting on the same array element.

The discussion that follows assumes there is only one producer and one consumer thread. This class is not designed to be used in other cases, when more than one thread is executing the Push() or Pop() functions.

The C++11 implementation of the RingBuffer class is defined in Listing 8.8. Consider the Pop() operation: if the buffer is empty, the operation fails, and that is the end of it. However, the Extract() member function keeps popping until the operation no longer fails, because an element has just been inserted at the target place. At this point, it is necessary to make sure the write to the buffer element by Push() *happens before* the read of the same buffer element by Pop(). Likewise, when the buffer is full, the read of the buffer element by Pop() must *happens before* the write to the same buffer element by Push().

```
template <typename T, int Size>
class RingBuffer
    {
    private:
     T buffer[Size];
     std::atomic<int> head, tail;
     std::atomic<int> C;

     int NextHead( int n)
```

Continued

```
    { if(n==(Size-1)) C++; return (n+1)%Size; }

 int NextTail( int n)
    { if(n==(Size-1)) C--; return (n+1)%Size; }
public:
 RingBuffer() : head(0), tail(0) {}

bool Push(const T& value)
   {
   int H = head.load(memory_order_relaxed);
   if(H == tail.load(memory_order_acquire)          // <=== T
           && C.load(memory_order_relaxed)) return false;
   buffer[H] = value;
   head.store(NextHead(H), memory_order_release);   // <=== H
   return true;
   }

bool Pop(T& value)
   {
   int T = tail.load(memory_order_relaxed);
   if(T == head.load(memory_order_acquire)          // <=== H
           && !C.load(memory_order_relaxed)) return false;
   value = buffer[T];
   tail.store(NextTail(T), memory_order_release);   // <=== T
   return true;
   }

void Insert(const T& value)
   { while(Push(value)==false); }

void Extract(T& value)
   { while(Pop(value)==false); }
};
```

LISTING 8.8

RingBuffer class

The Push() and Pop() functions are listed above. Imagine, for example, that the buffer is full, and that Push() is trying unsuccessfully to insert a new element, because it reads that head=tail and C!=0. Then, the consumer thread pops a value, and stores the next value of tail with release semantics at the place tagged <=== T in Listing 8.8. The thread executing Push() reads, again at the place tagged <=== T, the new value of tail with acquire semantics, which indicates that the buffer is no longer full. At this point, the load-store of the same value of tail with acquire-release semantics establishes the required "happen before" relation between the read of buffer[n] preceding the store and the write of buffer[n] following the load. This implements the correct synchronization for a concurrent operation on a full buffer.

It is easy to check that the correct synchronization for an empty buffer is enforced by the stores and loads of head at the places tagged as <=== H in Listing 8.8. In this case, the write of buffer[n] "happens

before" the read of buffer[n]. All the other atomic operations—initial reads of head and tail, reads and writes on C—are not involved in synchronizations, and are executed with relaxed semantics.

Example: TRBuff.C

The C++11 implementation of the RingBuffer class is tested by compiling the TRBuff.C with the CPP11_ENV=1 environment variable. Indeed, this portable class is also implemented in Windows—using the Windows atomic facilities discussed in Section 8.6—and in Pthreads—using the basic synchronization utilities due to the lack of atomic support. The producer thread inserts successive positive integer values in the buffer, followed by a negative value to inform the consumer that the game is over. The consumer keeps calling Extract() and printing the value until it extracts a negative value.

Example 4: TRBuff.C

Using a circular buffer to implement a producer-consumer pattern between two threads. To compile, run make trbuff.

8.6 TBB ATOMIC<T> CLASS

The tbb::atomic<T> class is similar to the std::atomic<T> class just examined, and the official documentation can be found in the "Synchronization" topic in the Reference Guide [17]. There are, however, a few differences.

In the tbb::atomic<T> class the template argument T is restricted to either an integer type (signed or unsigned integers or longs), to a pointer type, or to an enumeration type. Explicit atomic operations on integer types can be therefore programmed in a simple way, and atomic pointers can be used to implement thread-safe operations for other basic or user-defined data types, as the previous example has shown.

Listing 8.9 shows the properties of the tbb::atomic<T> class. The same conventions are used: capital letters—like X, Y—are type T atomic data items, while lowercase letters—like x, y—are ordinary type T data items.

```
// Variable declarations
// – – – – – – – – – – –
tbb::atomic<T> X;           // default constructor
tbb::atomic<T> X(T value);  // not atomic, see comments below

T  x, y;                // x, y are ordinary integers or pointers

// Three basic member functions, that all return the
// old value of X:
// – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
x = X.fetch_and_store(y)      // Execute X=y
x = X.fetch_and_add(y)        // Execute X+=y
```

Continued

```
x = X.compare_and_swap(y, z)  // If X equals z, execute X=y.
// Overloaded operators on atomic variables,
// Special cases of fetch_and_store, fetch_and_add
// – – – – – – – – – – – – – – – – – – – – – – – – – – –
x = X;          // read in x value of X – This is a load
X = x;          // write to X the value x – This is a store

X++;            // increment atomic variable
X––;            // decrements atomic variable
X+=x;           // add x to X
X–=x;           // subtract x from X
```

LISTING 8.9

Atomic operations in tbb::atomic<T>

This class provides basically the same services as the C++11 atomic types:

- There are no explicit load() and store() member functions, but these operations can be performed with X=x and x=X.
- Another way to perform a load, to be used later on, is using x = X.fetch_and_add(0), which does not modify X and returns its value.
- fetch_and_store() corresponds to exchange() in C++11.
- compare_and_swap() corresponds to compare_and_exchange() in C++11.
- fetch_and_add() implements the basic arithmetic operations on integers and pointers. As in C++11, atomic operations on pointers implement pointer arithmetic.
- There are no logical atomic operations.

The basic constructor is a default constructor, provided by the compiler. The C++ initialization rules for classes without explicit constructors apply here. The atomic data item is initialized to 0 (if integer) or NULL (if pointer) when declared as a global variable with file scope, as static data item inside a function, or as a data member in a class. There are some reasons for this, which are discussed in the documentation annex of [14]. The second constructor exposed in Listing 8.9, providing an explicit initialization value to the atomic variable, *is not atomic*. An alternative is to construct first the atomic variable with the default constructor and then load its value.

8.6.1 MEMORY ORDERING OPTIONS

TBB has a simpler scheme for memory ordering options, while maintaining the essential features of C++11. A memory_semantics enumeration is defined, providing the basic options for the acquire-release memory model. Memory semantic options *are not* passed as arguments to the member functions. Rather, template versions of the member functions are defined as having the memory_semantics values as template parameters, as shown in Listing 8.10.

```
namespece tbb
   {
   enum memory_semantics
       { acquire, release };
   }
```

```
// Enforcing acquire semantics on a store
// - - - - - - - - - - - - - - - - - - - - -
x = X.fetch\_and\_store<acquire>(y);
```

LISTING 8.10

TBB memory ordering options

The complete pattern is the following:

- All the member functions in Listing 8.9 have a sequential consistency memory order semantics.
- If the memory semantics have to be downgraded to acquire or release, the template version of the member function is used, as shown in Listing 8.10.
- The overloaded operators in Listing 8.9 all have an intrinsic memory semantics choice:
  - The store operation X=x uses *release* semantics.
  - The load operation x=X uses *acquire* semantics.
  - The three other basic functions use *sequential consistent* semantics.

A comment is useful at this point. It was observed before that there are no explicit load or store operations, which these can be implemented with x=X (with acquire semantics) or X=x (with release semantics). If, instead, sequential consistent semantics is required for a load or a store, the fetch_and_store(x) function can be used for a store, and x = fetch_and_add(0) can be used for a load.

Example: TRBuff_T.C

The C++ classes and examples discussed before can easily be ported to the tbb::atomic<T> class. Notice, in particular:

- The class RingBuff_T implements the RingBuffer class using the TBB atomic class. The class is defined in the header file RBuff_Tbb.h.
- This class is tested in the TRBuff_T.C code.

Example 5: TRBuff_T.C

Producer-consumer pattern between two threads, TBB implementation. To compile, run make trbuff_t.

8.7 WINDOWS ATOMIC SERVICES

The Windows API, being a C library, does not defines atomic types. But Windows proposes a large number of functions—called Interlocked functions—implementing the mutual exclusion or synchronization services discussed before, including the memory order options. They are well documented in [13].

Windows API -> System services -> Synchronization -> Interlocked functions

This reference shows a long list of functions, with a brief description of their action. Clicking on the function name, another web page is reached with a more detailed description of the function signature and a number of comments on their correct usage. All the interlocked functions follow a number of rules concerning the function names and signatures:

- Atomic operations act on char, integer, or void pointer types.
- The atomic operation, the target data type, and the memory order option are all coded *in the function name*. This is the reason there is an impressive number of interlocked functions.
- Function names are of the form:
  - Interlocked(Operation)(Data size)(Memory option)
  - (Data size) and (Memory option) are optional. Default values are for target data a 32 bit long, and for memory option sequential consistency.
  - Explicit data sizes are 8 for char, 16 for short int, and 64 for long long int.
  - Explicit memory options can be Acquire, Release to implement the acquire-release memory model, and NoFence, which obviously corresponds to the relaxed model discussed before for C++11.
- Function arguments are in all cases the target data item—an in-out argument passed by address—and additional data items needed for the operation, passed by value.
- The return value is in all cases either the previous or the last value of the target data.

We have stated that the default memory order option is sequentially consistent. In fact, the Windows documentation guarantees that this default option implements a strong acquire-release memory fence. Notice that atomic operations only act on void pointers: the address of the target data item is passed cast as a void*. Indeed, this is the only way of manipulating in C a generic address. Another piece of information to keep in mind is that some interlocked functions—like CompareExchange—require that target data items be 32- or 64-bit aligned, otherwise they may behave in an unpredictable way on multicore x86 platforms or any other non-x86 platform.

Porting std::atomic code to Windows native code is perhaps a bit tedious but not difficult, since the basic concepts and ideas are the same. In order to illustrate the usage of the interlocked functions, the Windows implementation of the RingBuffer class is listed below.

```
template <typename T, int Size>
class RingBuffer
    {
    private:
     T buffer[Size];
     long head, tail, C;

     int NextHead( int n)
         {
         if(n==(Size-1)) InterlockedIncrementNoFence(&C);
         return (n+1)%Size;
         }

     int NextTail( int n)
         {
         if(n==(Size-1)) InterlockedDecrementNoFence(&C);
```

```
      return (n+1)%Size;
      }

 public:
  RingBuffer() : head(0), tail(0), C(0) {}

 bool Push(const T& value)
    {
    int H = head;
    if(H == InterlockedAddRelease(&tail, 0) && (C>0)) return false;
    buffer[H] = value;
    InterlockedExchangeAcquire(&head, NextHead(H));
    return true;
    }

 bool Pop(T& value)
    {
    int T = tail;
    if(T == InterlockedAddRelease(\head, 0) && (C==0)) return false;
    value = buffer[T];
    InterlockedExchangeAcquire(&tail, NextTail(T));
    return true;
    }

// Insert() and Extract() do not change
};
```

LISTING 8.11

Windows implementation of the RingBuffer class

The head and tail indices, as well as the counter C, are now ordinary long integers. To atomically increment the C counter the InterlockedIncrementNoFence() function is used, with a relaxed memory option because this atomic operation is not involved in a synchronization pattern. The same applies to the atomic decrements of C.

In the Push() and Pop() head and tail are stored with acquire semantics and loaded with release semantics. The stores are straightforward: the store operation is performed by InterlockedExchange(), which atomically replaces the value of the first argument by the value provided by the second argument. Incidentally, to check the consistency with previous discussions it is possible to verify that there is no InterlockedExchangeRelease() function: store operations accept only acquire semantics.

The load operation is slightly more subtle. There is no explicit load atomic operation, like in C++11, because the target is a standard data item that can be read directly. But an atomic operation returning the target value with release semantics is needed, to implement the acquire-release synchronization. The InterlockedAdd(&target, data) function adds data to the target and returns the new value. Then, just adding 0 atomically returns the data value. This is what is done in Listing 8.11, by choosing the function with release semantics.

The circular buffer implementation provided by this class operates correctly, as shown by the same test code shown before in Example 4, TRBuff.C, but compiled in the Windows environment.

Example 5bis: TRBuff.C

Using a Windows native circular buffer to connect two threads, when compiled in the Windows environment. To compile, run nmake trbuff.

It is also useful to look at the SpinLock.h source where the Windows implementation of this class is defined.

8.8 SUMMARY

Atomic services have been used in this chapter to implement synchronization utilities—lock-free reduction services, spin locks, and ring buffers—incorporated in the vath library with the standard portable strategy: the same programming interfaces in all programming environments. Their status is summarized below:

- **SpinLock**: Programming interface as in Listing 8.7.
  - Implemented in C++11 and Windows using the corresponding atomic services.
  - In Pthreads, there are no atomic services. This class just encapsulates the native spinlock mutex proposed by Pthreads.
- **AReduction**: Same programming interface as Reduction.h, described in Chapter 5.
  - Implemented in C++11 and Windows using the corresponding atomic services.
  - No Pthreads implementation.
- **RingBuffer**: Programming interface as in Listing 8.8.
  - Implemented in C++11 and Windows using the corresponding atomic services.
  - Implemented in Pthreads using the basic synchronization primitives.
- **RingBuff_T**: Programming interface as in Listing 8.8. A TBB implementation of RingBuffer.