CHAPTER

HIGH-LEVEL SYNCHRONIZATION TOOLS

9

9.1 INTRODUCTION AND OVERVIEW

This chapter examines several high-level concurrency patterns common in applications programming, and the C++ utility classes contained in the vath library proposed to manage them. Our pedagogical purpose is to discuss a variety of examples illustrating various ways of synchronizing threads. The vath utility classes sometimes simply encapsulate basic synchronization primitives in portable interfaces, and often propose other services that are not directly available in the basic multithreading libraries discussed in this book.

The implementation of the vath synchronization utilities is not discussed in detail. References to implementations are made only when a relevant pedagogical issue deserves particular attention. Interested readers can consult the vath source code, which is well documented. Most of the C++ synchronization classes are rather simple, relying on aspects of the basic libraries discussed in this or in the previous chapters. Effort has been made to have sources as simple and readable as possible. Readers willing to develop a deeper understanding of these subjects can also consult several books that present the state-of-the-art in developing of custom synchronization utilities. A rigorous and advanced discussion is contained in *The Art of Multiprocessor Programming*, by M. Herlijy and N. Shavit [25]. Advanced material in the C++11 environment is available in [14].

The vath services are discussed in detail, for several reasons:

Extended portability

This simple library has a Pthreads, Windows, and C++11 implementation, with common programming interfaces. It can be used as such in any Unix-Linux or Windows system.

Interoperability with OpenMP

This library proposes optional thread management interfaces (thread pools). But the synchronization utilities can be used, for example, in an OpenMP environment, as is the case for several examples in the following chapters.

Pure OpenMP utilities

Some of the utilities dealing with synchronizations among individual threads have a fourth implementation (besides Pthreads, Windows, and C++11) using pure OpenMP code. This option has been included for OpenMP developers willing to avoid interoperability concerns.

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00009-5

The vath library deals with thread management and synchronization. This chapter is devoted to thread synchronization. A lightweight thread management utility—the SPool class—has already been introduced in Chapter 3. A more sophisticated thread management utility called NPool, exhibiting many of the OpenMP and TBB features, is discussed in detail in Chapter 12. After discussing the vath thread synchronization utilities, this chapter concludes with a general overview of thread pools as high-level thread management utilities.

9.2 GENERAL COMMENTS ON HIGH-LEVEL SYNCHRONIZATION TOOLS

As stated in Chapter 2, threads are introduced to run concurrent tasks, implementing mandatory parallelism. In all the examples examined so far, a worker thread was launched for each parallel task in the application, thereby establishing a one-to-one mapping between tasks and threads. This strategy will henceforth be referred to as a *thread centric* programming paradigm.

This is not, however, the only way in which multithreaded environments like OpenMP or TBB are structured today. Starting in the next chapter, *thread pools* will be introduced as high-level thread management environments, where threads are created first, wait silently in a blocked state if they are not needed, and are woken up and activated to perform tasks if and when their participation is required. In this environment, it is always possible to activate one thread for each specific task in the application, but it is also possible to run applications involving more parallel tasks than threads in the worker team. A *task-centric* execution model can be implemented, in which tasks are queued and executed when worker threads become available. Even better, they are not necessarily executed in the order in which they were submitted: the task scheduler may be unfair, for very good reasons that will be discussed later. In this context, the task to thread mapping is not necessarily one-to-one. Parallel computations may engage a number of parallel tasks substantially bigger than the number of worker threads.

Figure 9.1 shows the most general parallel execution context: tasks are executed by threads, and threads run on cores. The ideal context is clearly the one in which there is a one-to-one mapping

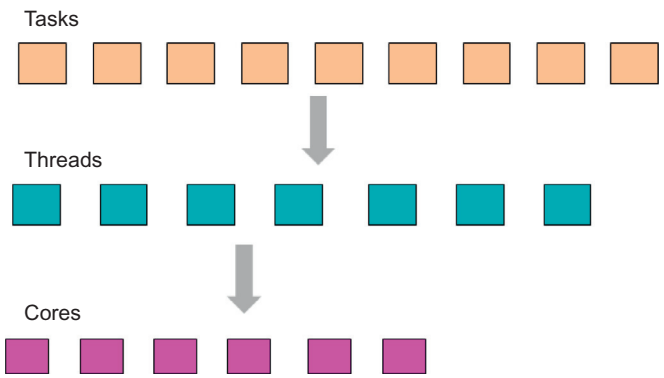

FIGURE 9.1

Tasks are executed by threads, and threads run on cores.

between tasks, threads, and cores. However, this is often not the case. It may happen that the total number of tasks is not hardwired in a computation, because tasks launch internally new tasks. It may also happen that the number of executing cores is less than the number of active threads in the application.

The purpose of this chapter is the synchronization of threads. The synchronization algorithms are programmed in the tasks executed by the threads. However, it is important to keep in mind that:

Synchronization tools synchronize threads, not tasks. Using synchronization tools without clear control of how tasks are mapped to threads can sometimes deadlock the code or deliver incorrect results. This issue will be frequently reexamined in the following chapters.

A simple example helps to clarify this observation. Let us assume a program involving a parallel computation on N tasks, with a barrier synchronization among them. The barrier utility is naturally initialized to N participants. The code deadlocks when run on M threads (M<N), because at the barrier synchronization point it will keep waiting for threads that will never come. Another example was given at the end of Chapter 4, concerning thread local storage. Thread safety is guaranteed only if the tasks that access thread private variables are assigned to different, well-identified threads.

Also keep in mind that only the mapping of task to threads is delicate, the mapping of threads to cores is not. If a number of correctly synchronized threads is running on a smaller number of cores, threads will be preempted and rescheduled by the operating system, but the synchronizations will in all cases operate correctly, and the application should run flawlessly, with lower performance.

9.3 OVERVIEW OF THE VATH SYNCHRONIZATION UTILITIES

There are several synchronization utilities discussed in this chapter:

- **Timers**, objects that put a thread in an idle wait state for a predefined amount of time.
- **Boolean locks**, objects that encapsulates the basic idle or spin waits for an event, discussed in Chapters 7 and 8.
- **Synchronization points**, a rendezvous point for two or more threads that synchronizes a write and one or several reads of a memory location. This is a safe and robust way of exchanging data between a producer thread (the one that writes to memory) and one or more consumer threads that read data values from memory.
- **Barriers**, synchronization tools that establish a rendezvous point for a predefined number of threads, discussed in Chapter 7.
- **Blocking barriers**, barriers in which the waiting threads are not released by the last thread reaching the barrier point. They are released by an external client thread.
- **Thread safe queue**, a first-in, first-out *thread safe* queue container implementing the producer-consumer paradigm. Consumer threads wait for data if the queue is empty, and producer threads wait if they attempt to insert data in a full queue. The queue capacity is chosen by the programmer.

- **Pipelines**, tools that encapsulate the synchronization of pipelined threads, thereby simplifying the programming of applications requiring this specific concurrency pattern. Pipeline classes are discussed in Chapter 15, entirely devoted to this issue.
- **Reader-Writer locks**, generalized locks that enforce mutual exclusion for some operations only (writes), while other operations (reads) may proceed concurrently in a standard way.

The discussions of these utilities describe their purpose and their user interface, followed by examples. Occasionally, qualitative indications about the implementation are given, when useful for a better understanding of their operation.

9.3.1 CODE ORGANIZATION

The vath utilities are C++ classes, providing services to C-C++ client code. They only require from the user some very limited basic knowledge on *programming with objects*. The C++ source code relies, of course, on the Standard Template Library (STL), but this is hidden to the end user. Sometimes, the *generic programming* C++ features are used, with the purpose of writing generic code—template functions and classes—in which one or several data types are kept as parameters to be specified by the client code later on, at compilation time. For example, class templates are used in utilities where data is exchanged between threads, leaving as a template parameter the exchanged data type.

With the only exception of Timers (which concern only one thread) all synchronization objects must naturally be shared by several threads. Therefore, they have to be declared with global scope, and there are in this case the two options we already discussed for the SPool utility in Chapter 3. If the parameters needed by the object constructor are known at compile time, synchronization objects can be declared directly as global objects. Otherwise, a delayed construction of the object must be adopted: a global pointer to the object is declared, initialized later on by a call to new.

```
thread_function(void *P)
   {
   Timer T;
   ...
   T.wait(200);     // wait for 200 milliseconds
   ...
   }
```

LISTING 9.1

Using a proprietary Timer.

9.4 TIMERS

The Timer utility puts the caller thread to wait in a blocked state for a predefined duration time interval (coded in milliseconds). It was indicated in Chapter 3 that C++11 already has a Timer provided by the function std::this_thread::sleep_for(), and that Windows also provides a built-in Timer through the Sleep() function. In Pthreads, however, programming a Timer is substantially more involved. This class was introduced mainly to be able to write portable application code.

The interest of Timer objects is the capability of delaying for a given time interval the execution flow of a thread. The first example given in Chapter 3 simulated a lengthy I/O operation by getting a thread out of the way for a few seconds. When testing synchronization constructs in several examples in the chapters that follow, different threads are delayed for very different time intervals in order to more efficiently track the way the synchronization pattern operates.

9.4.1 USING A TIMER

The Timer class is declared in Timer.h. Here is the public interface of the class:

TIMER PUBLIC INTERFACE

Timer()

- Constructor.
- Creates a Timer object.

Timer()

- Destructor.
- Destroys the Timer object.

int Wait(long ms)

- Caller threads waits for ms milliseconds.
- Returns 0 on success, or positive number on failure.

9.4.2 COMMENT ON IMPLEMENTATION

In Windows and C++11, this class just encapsulates the native timer service. In Pthreads, a thread calling this function performs a *timed wait* for a predetermined time interval, on a condition variable totally hidden from any other thread in the process, and that consequently will never be signaled. *The timed wait will therefore be necessarily timed out after the requested wait interval*. It is therefore important to avoid sharing Timers between threads.

9.5 BOOLEAN LOCKS

BLock (Boolean Lock) objects are synchronization objects that encapsulate in a portable and user friendly way the basic *idle wait on condition* event synchronization mechanism, discussed in Chapter 6, or the *spin wait protocol* discussed in Chapter 7. A Boolean lock can be seen as a black box containing internally a Boolean state variable (a predicate) taking the values (true, false), as well as a guarding mutex and—for idle waits only—a condition variable signaling the changes of state. The expected synchronization event corresponds to the fact that the state variable is toggled.

Client treads can do three different things with the Boolean lock:

- They may read or reset the value of the internal state variable.
- They may reset the state variable and notify waiting threads that it has been changed. For spin waits, this notification is not needed, because waiting threads keep reading the state variable and can detect its change.

- They may wait until some other thread changes the state variable. Idle waits can be *timed waits* for a fixed numbers of milliseconds, or indefinite waits until the state change occurs. Spin waits are, naturally, indefinite waits.

9.5.1 USING THE BLock CLASS

The BLock class is the general-purpose utility that encapsulates the idle wait protocol discussed in Chapter 6. A typical *use case* for the Boolean lock is the IO example discussed in Chapters 3 and 6. When a thread is launched to perform an assignment—like a computation, an important data transfer on the network or a lengthy I/O operation to a file—this utility enables the main thread or other threads to check every now and then, using timed waits, if the operation has been completed, or simply wait indefinitely until it is done. The synchronization pattern proceeds as follows:

- The main thread sets the Boolean lock to false before launching the worker thread.
- The worker thread does its job and, when it is finished, sets the Boolean lock to true and notifies the change.
- The main thread, at some point, waits for the Boolean lock to switch to the true state.

Here is the description of the BLock public interface:

BLock INTERFACES

BLock()

– Constructor.
– Creates a Boolean Lock with the default internal state (false).

BLock(bool b)

– Constructor.
– Creates a Boolean Lock with an internal state equal to b.

bool GetState()

– Returns the value of the internal state.

void SetState(bool b)

– Sets the internal state to the value b.

int Wait_Until(true/false, long timeout)

– Caller thread waits until the internal state becomes true or false for *timeout* milliseconds.
– If *timeout=0*, this becomes an unconditional wait, and the caller thread waits forever until the true state is notified.
– Return value: 0 if the function returns because true has been notified, and 1 if the function returns because the wait has been timed out.

void Set_And_Notify(bool new_state)

– Sets the internal state to new_state and signals the change.
– Only one waiting thread is woken up.

void Set_And_Notify_All(bool new_state)

– Sets the internal state to new_state and broadcasts the change.
– All waiting thread are waken up.

We insist on the fact that the client code that puts a thread to wait may choose between an *unconditional wait*—until the condition is notified—or a *timed wait*, in which case threads do not wait forever: the wait on condition returns after a predetermined time interval if the condition is not notified. The client thread can return to do some other useful work before waiting again.

In this class, idle waits are timed for a specified number of milliseconds passed as argument in the function call. For an unconditional wait, until the event is signaled, the timewait argument must be 0 milliseconds.

9.5.2 BLock EXAMPLES

The IO example discussed in Chapter 6, showing the way condition variables operate, are re-examined next. The synchronization pattern is the following:

- The master thread sets the private BLock predicate to false and launches the IO thread.
- The IO thread waits for a long time interval (5 s), toggles the private predicate to true, and notifies the event.
- After launching the worker thread, the main thread starts a do loop in which it keeps waiting for the true state for only 1 second. If the wait is timed out, the master thread prints a message, starts a new loop iteration, and waits again. The loop breaks when finally the timed wait ends not because it has been timed out, but because the change of state has been notified.

Here is the listing of the source file TBlock.C.

```
SPool TH(1);      // set of one worker thread
BLock *B;

void th_fct(void *arg)
   {
   Timer T;
   T.Wait(5000);
   B->SetAndNotify(true);
   std::cout << "\n Boolean flag set and notified" << std::endl;
   }

int main(int argc, char **argv)
  {
  int status;
  B = new BLock(false);     // create and initialize BLock object
  TH.Dispatch(th_fct, NULL);     // launch worker thread
  do
        {
        status = B->Wait_Until(true, 1000);
        std::cout << "\n Got return value = " << status
                  << std::endl;
        } while(status==0);
```

Continued

```
    TH.WaitForIdle();
    delete B;               // delete BLock object
    return 0;
    }
```

LISTING 9.2

TBlock.C code.

Example 1: TBlock.C

To compile, run make tblock. Running the program with the parameters in Listing 9.2 shows that the main thread prints 4-5 timeout messages, as expected, before detecting the change of state.

There is a second example, in source file TBlockOmp.C, which shows OpenMP interoperability. The code is the same as in the previous case, except that the parallel directive in an OpenMP environment is used, instead of the SPool utility, to run the worker threads. Incidentally, keep in mind that this is the *only* way of implementing timed waits in an OpenMP context.

Example 2: TBlockOmp.C

To compile, run make tblockomp. The same output as in the previous example.

9.5.3 BOOLEAN LOCKS IMPLEMENTING SPIN WAITS

The Boolean Lock utility just discussed is a general-purpose utility with very acceptable efficiency in most cases. There are, however, computational contexts with high synchronization overhead where Boolean locks are intensively used for very short unconditional waits, and in these cases the fact that waiting threads are systematically preempted and rescheduled may have a negative impact on performance. In these contexts, spin waits are more efficient. There are two spin wait Boolean lock classes provided by the vath library. They are portable, and can be used "as such" in any programming context.

- **SpBlock**: Implements the spin wait protocol discussed in Chapter 7, where waiting threads keep reading a shared state variable, protected by the SpinLock object discussed in Chapter 8. Implementations in Pthreads and Windows-C++11 are very different:
  - In Pthreads, the SpinLock encapsulates the native spinlock mutex.
  - In C++11-Windows, the SpinLock is constructed using atomic services, as discussed in Chapter 8.
- **OBlock**: The spin wait protocol is implemented using OpenMP locks to protect the state variable. This is OpenMP code; this class is proposed for users that prefer to avoid interoperability issues.

It goes without saying that, in cases of very high synchronization contention, the atomic implementations of SpBlock are the most efficient ones, as the examples in Chapter 14 will show. *These classes have all the same public interfaces*, which is the same as the BLock interface, with only two differences:

- There are no timed waits, and the Wait_Until(true/false) member function does do not receive a timeout argument. They wait forever until the expected change of state occurs.
- There are no Set_And_Notify() or Set_And_NotifyAll() functions. Indeed, waiting threads are always active, constantly checking the state variable, and they will read the new value as soon as it becomes visible to them.

Example. The previous example is now coded using the OBLock class, except that, since there is no timed wait available, the master thread just waits for the predicate to be toggled. The source file is TOmpBlock.C.

Example 3: TOmpBlock.C

To compile, run make tompblock. Pure OpenMP code.

9.6 SynchP< *T* > TEMPLATE CLASS

SynchP objects are special-purpose Boolean locks used to order a write and one or several reads of a generic data item by different threads.

Consider the case where thread A (henceforth, called the producer thread) needs to communicate some data value to threads B, C (henceforth, called the consumer threads). The producer thread can, of course, write the data to some global data buffer, and the consumer threads can read from it, but it is obvious that a synchronization is required to make sure the write took place before the reads, *and that the reading threads are reading the last value written to the data item, as discussed in Chapter 7*.

To handle this pattern, an object is constructed, analogous to a Boolean lock, but also incorporating the exchanged data item as an internal private variable. Since the data type that will be accessed is not known in advance, the *generic programming* capability of C++ is used to keep it as a *template parameter* to be specified by the client code later on.

9.6.1 USING SynchP<T> OBJECTS

The SynchP.h file contains the SynchP<T> template class, where T is the symbol for an abstract data type that is exchanged between the producer and consumer threads. Remember that template classes are declared *and defined* in the included *.h file. There is no separate object file incorporated in the library, simply because the compiler cannot generate object code before the client code defines what the data type T is.

The programming interface is very simple. Besides the constructor and the destructor, there are only two methods, Post() called by the producer thread, and Get() called by the consumer threads.

SYNCHP<T> PUBLIC INTERFACE

SynchP<T>()

– Constructor.
– Creates a SynchPoint<T> object with the default internal state (false).

void Post(T& elem, int nReaders)

- Posts the T data item of value *elem*.
- *nReaders* is the number of threads that will read the data.
- Function call waits eventually for the object to be ready.
- Called by the producer thread.

T Get(T)

- Reads and returns the previously posted T data item of value *elem*.
- Function call may wait if producer is late.
- Called by the consumer thread.

The synchronization pattern implemented by this interface, illustrated in Figure 9.2, proceeds as follows:

- The initial state of the Boolean state variable is false. When the state on the SynchP object is false, the object is either brand new, or it is available to be reused again because any previous data transfer has already been completed by all the planned reader threads.
- The producer thread calls the Put() function to write the data by indicating also how many readers are expected. If this function finds a true state, it concludes that the SynchP object is still busy with the previous transfer and waits for false before proceeding. As soon as the state is false, the producer thread writes the data, toggles the state to true (because the object is now busy again!), notifies the change to consumer threads that may be waiting for data, and returns.
- Notice that the producer thread is only blocked if the SynchP object is not ready. Otherwise, *the producer thread is not blocked*.
- Consumer threads eventually wait for true if the producer thread is late. Then they read the data and decrease an internal counter that keeps track of the number of reads. When the read counter reaches 0, the state is changed to false (object is now available for reuse) and the change is signaled.

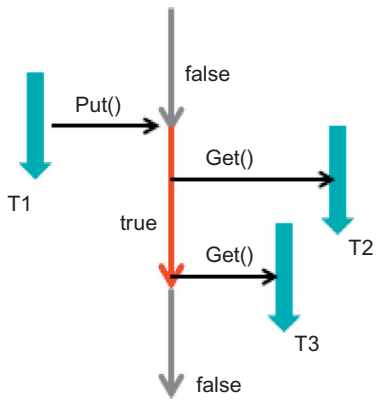

FIGURE 9.2

SynchP operation: one writer and two readers.

- *Notice that consumer threads are only blocked if the producer is late*. Otherwise, they are never blocked.

9.6.2 COMMENTS

This is a flexible synchronization mechanism in which threads are forced to wait only if it is absolutely necessary. Think of what happens if the only available synchronization mechanism is a barrier, as is the case in pure OpenMP. The OpenMP barrier synchronizes *all* the threads—the producer, the consumers, and all the remaining worker threads—at the barrier synchronization point. Then, it is sufficient to put the write of the producer and the reads of the consumers before and after the barrier call, respectively. However, if the operation needs to be repeated, a new barrier call is required *before* the next write to make sure all the previous reads are finished. Therefore, we have to count *two barrier calls*, each one blocking all threads, for each point-to-point data transfer operation. The synchronization provided by the SynchP class is much more efficient. When running 248 threads in a Intel Xeon Phi, it is totally wasteful to block the whole team just to exchange data between a few of them.

The template code for the SynchP class listed above is very general, because it applies to *any* data type (and not only to the primitive data types like double, integer, etc.), provided that the class defining the data type T: satisfies some requirements.

- The class has a no parameter constructor—so that the compiler can implement the instruction `T retval = T()`.
- The class has an assignment operator, so that the compiler can implement the instruction `retval = element`.

9.6.3 SynchP<T> EXAMPLES

The example that follows proposes code in which one thread repeatedly posts a double to two other threads (see Figure 9.2). The source file is TSynch.C. The master thread proceeds as follows:.

- First, it launches the two consumer threads, and waits for 3 s.
- Next, it posts a double value to two consumer threads. A new double value is posted next, by reusing the synchronization object.
- Finally, it joins the consumer threads.

As far as the consumer threads are concerned, they wait twice for 500 ms and try to read the next value coming from the master thread. Obviously, they will have to wait. Here is the code source listing.

```
SPool *TH;
SynchP<double> B;

void *th_fct(void *arg)
    {
    double d;
    int rank = TH->GetRank();
    Timer X;
```

Continued

```
    for(int n=0; n<2; n++)
       {
       X.Wait(500);
       d = B.Get();
       std::cout << "\nWorker thread " << rank << " got value " << d
                 << std::endl;
       }
    }

int main(int argc, char **argv)
    {
    int nTh;
    Timer Tm;

    if (argc==2) nTh = atoi(argv[1]);
    else nTh = 2;
    TH = new SPool(nTh);

    // launch worker threads
    // - - - - - - - - - - -
    TH->Dispatch(th_fct, NULL);

    // Main thread code
    // - - - - - - - - - -
    Tm.Wait(3000);
    double d = 1.3546;
    for(int n=0; n<2; n++)
       {
       d += 1.0;
       B.Post(d, nTh);
       std::cout << "\n Main : value posted" << std::endl;
       }

    TH->WaitForIdle();
    std::cout << "\n Main :  worker threads joined" << std::endl;
    return 0;
    }
```

LISTING 9.3

TSynch.C

This example also shows how to instantiate the SynchP<T> class for a specific data type (in this case a double): it is sufficient to declare a global SynchP<double> object. From there on, the compiler generates code that is a specialization of the template code to the particular data type required in this application.

Example 4: TSynch.C

To compile, run make tsynch. By modifying the Timer calls the order in which events happen can be inverted in order to check that the synchronization pattern works in all cases.

There is also in this case a second example, in source file TSynchOmp.C, with the worker threads managed in an OpenMP environment, to validate the interoperability with OpenMP.

Example 5: TSynchOmp.C

To compile, run make tsynchomp.

9.6.4 OpenMP VERSION: THE OSynchP< *T* > CLASS

This class is yet another *pure OpenMP utility* that performs exactly like the SynchP<T> class by using OpenMP busy waits. The public interfaces are exactly the same.

Example 6: TOmpSync.C

To compile, run make tompsynch.

9.7 IDLE AND SPIN BARRIERS

Barrier synchronization is probably one of the most commonly used synchronization patterns in application programming, particularly in scientific codes. At a barrier synchronization point, members of a set of N cooperating threads wait until all the members are present. A full discussion is given in Chapter 6 concerning the way this utility operates, as well as the subtleties involved in programming a reusable barrier object. To the extent that the barrier algorithm is based on condition variables, the threads waiting in a barrier call are executing an *idle wait*.

The vath library proposes four barrier synchronization classes:

- The Barrier class, implemented as discussed in Chapter 6, by having the waiting threads performing an idle wait. This Barrier class is to some extent redundant because Pthreads and Windows have today a barrier utility. We use it in the examples in order to have portable codes.
- The SpBarrier class, which is a *spin barrier* where the waiting threads keep spinning in user space without releasing the CPU. Its implementation relies on the SpinLock mutex class discussed in Chapter 8.
- The ABarrier class, which is, again, a *spin barrier*, but using a highly efficient barrier algorithm based on an atomic predicate and, curiously, the usage of thread local storage, taken from [25]. Its implementation in C++11 and Windows relies on the native support for atomic variables and thread local storage. *There is no Pthreads implementation, due to the absence of atomic support*.
- The TBarrier class, identical to ABarrier, but using TBB atomic and thread local storage utilities. This redundancy is introduced because the ABarrier class does not run in Pthreads (due to the

absence of atomic support). The TBB implementation is fully portable, and can also be used in a Pthreads environment.

9.7.1 USING THE BARRIER CLASSES

The public interfaces are the same for all kinds of barrier objects, so that it is quite easy to substitute them in an application, with one minor exception for the ABarrier case, discussed in the next subsection:

BARRIER PUBLIC INTERFACE

Barrier(int N)

- Constructor.
- N is the number of worker threads synchronized by the barrier.

int Wait()

- Calling threads wait until the group of cooperating threads is released by the last thread arriving at the synchronization point.
- Returns a positive number if the call fails. If the call is successful, all cooperating threads return 0 except the last one, which returns (-1).

Why should the last thread reaching the barrier point return (-1) instead of 0? There is nothing special about the last thread; this is just an easy way of selecting one of the partner threads after the barrier call. Often a barrier is used to move from a *parallel region* where all the threads share some specific assignment to a *sequential region* where a treatment is trusted to only one thread. The barrier return value is a handy way of selecting with an if statement one specific active thread, in particular in cases where the worker threads do not have a built-in rank value identifier.

As far as performance is concerned, it is difficult to see the difference between the different barrier implementations in codes with small or moderate barrier contention. But in Chapter 13 an application where there is a substantial amount of barrier contention will be considered, and in this case the atomic spin barrier code exhibits much better performance than the code using the idle wait barriers.

9.7.2 ABarrier IMPLEMENTATION

The case of the ABarrier utility illustrates how custom synchronization utilities, constructed using services provided by the basic libraries, often help to significantly improve the performance of applications exhibiting substantial synchronization overhead. Chapter 14 is partly devoted to an application dealing with the solution of two-dimensional differential equations. When examining the scaling behavior of this application, it was found that the vath implementation using the SPool environment and the standard Barrier class discussed above was very deceptive: starting from four threads, the wall execution time started to increase with the number of threads!

It soon became clear, after a few tests, that the SPool environment itself was not to blame, so the next step was to look for improved barrier algorithms. Scalable barrier algorithms have received sustained attention because of the necessity of efficiently synchronizing thousands or tens of thousands of MPI processes in large-scale distributed memory applications.

The ABarrier algorithm that follows is taken from Chapter 10 in *The Art of Multiprocessor Programming* [25]. It is shown in Figure 9.3. There are no condition variables, because this is a spin

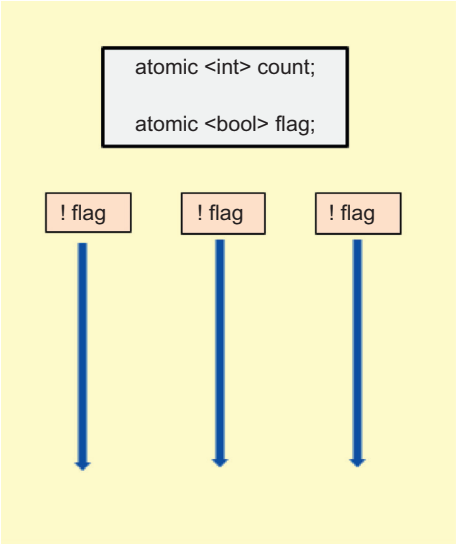

FIGURE 9.3

ABarrier algorithm.

barrier: waiting threads keep reading the predicate to detect the change. There is no mutex locking, because atomic variables are used. Besides std::atomic, the C++11-Windows implementations rely on the thread local storage services discussed in Chapter 4, because each thread keeps a thread private bool that preserves its value across successive Wait() function calls. This feature is needed to make the barrier algorithm reusable (remember the barrier discussion in Chapter 7).

The ABarrier object has two private atomic data items: an integer count that is initialized to the number of participating threads, and decreased every time a new thread starts a wait. There is also an atomic bool flag, a predicate that is toggled to release the waiting threads, initialized to an arbitrary value, say, false. The participating threads start initially with a thread private flag opposite to the predicate. Now, the algorithm operates as follows:

- A thread that calls Wait() decreases the counter and starts a spin wait comparing its private flag to the predicate, until they are equal.
- The last thread to call Wait() resets the counter to the number of threads, and toggles the predicate. This releases the remaining waiting threads.
- Before returning from the Wait() call, *all threads toggle their private flag*, so the initial condition is restored (private flags opposite to the predicate) and the barrier can be used again.

The source code is in ABarrier.h. This class in implemented in C++11 and Windows, and a small modification is introduced in the user interface to dispose of a portable utility. The point is that, as discussed in Chapter 4, the thread local storage services in C++ and Windows have some differences. All threads must initialize their thread local variables, and in C++11 this is automatic the first time each thread calls the function using them. But in Windows the initialization is not automatic, and must

be done explicitly before the thread local variable is first used. Therefore, an initialization function is added to the member functions listed below:

void InitTLS() is called by each thread before any call to the Wait() function. In C++11, this function does nothing. In Windows, it is relevant.

void InitTLS() is called by each thread before any call to the Wait() function. In C++11, this function does nothing. In Windows, it is relevant.

A real-life test of this class will be discussed in Chapter 14. Just to check that it works, a simple example is proposed next in which a team or worker thread successively prints messages to stdout and performs a barrier call. The default number of threads is 4, but this can be overridden from the command line. Running the code shows that, no matter how many threads, the successive messages are correctly ordered.

Example 7: TABarrier.C

To compile, run make tabarrier. The number of threads (4 by default) can be changed from the command line.

Look in particular at the initialization issue in this example. Before launching the real parallel job, a preliminary parallel job is executed in which each thread calls the initialization function.

9.8 BLOCKING BARRIERS

In the standard Barrier utility, the participating threads are blocked at the barrier synchronization point, and they are released by the last thread performing the Wait() call. This is a tool for internal synchronization of a team of *N* working threads.

A blocking barrier implements a similar synchronization pattern, except that the participant threads *are not released* by the last one that arrives to the synchronization point: they remain blocked. They are all released when an external client thread calls the ReleaseThreads() member function, as sketched in Figure 9.4. In fact, this utility introduces a synchronization between a team of worker threads performing a parallel computation and a client master thread that is driving the whole process. This utility is useful when the worker threads need new results injected from the client thread to proceed beyond the barrier synchronization point, or when the client thread needs to recover some partial result at some intermediate point in the execution of a parallel job by the worker threads.

9.8.1 USING A BLOCKING BARRIER

A blocking barrier is a barrier, so they share with the barrier classes the same constructor and the same Wait() function called by the worker threads engaged in the barrier synchronization point. There are, however, three new member functions dealing with the synchronization with the master thread. These new member functions, called naturally by the master threads, behave as follows:

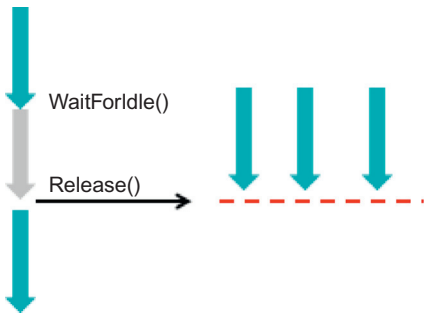

FIGURE 9.4

Blocking barrier operation.

ADDITIONAL BkBarrier PUBLIC INTERFACES

`void WaitForIdle()`

- Called by client threads.
- This function returns when all the participant threads have reached the barrier point.

`void ReleaseThreads()`

- Called by client threads, after a call to WaitForIdle().
- The participant threads are released.

`bool State()`

- Called by client threads.
- Returns true if all threads have reached the barrier point, 0 otherwise.

9.8.2 FIRST SIMPLE EXAMPLE

This section develops two examples. The first one is in source file TBkBarrier.C, a simple test of the BkBarrier class implementing the following scenario:

- The main thread launches a number of worker threads (read from the command line; the default is 2 threads).
- All the worker threads execute the same code:
  - they enter a loop in which they first wait for a random time interval,
  - then they print an identification message,
  - and finally they wait on a blocking barrier.
  - This loop is executed 4 times.
- The main thread enters a loop in which:
  - it first waits for the worker threads to be idle (so that they have all written their messages),
  - then it prints a line and releases the threads, so that they can continue their planned execution flow.
  - This loop is also executed 4 times.
- Finally, the main thread joins the worker threads.

The code is simple and compact, and it uses one or two simple tricks that are well documented in the source file. For this reason, it will not be discussed any further here. I suggest executing the code with a large number of threads (larger than the number of available cores in your system) to verify that the complex synchronizations used in this example work properly. You will see the lists of identification messages emitted by the worker threads, separated by the dotted line printed by main() at each blocking barrier synchronization point.

There are two versions of this example. In the first one, in source file TBkBarrier.C, worker threads are managed using our standard thread management utility SPool. In the second version, in source file TBkBarrierOmp.C, worker threads are managed by OpenMP.

Example 8: TBkBarrier.C

To compile, run make tbkb. The number of threads is read from the command line (the default is 2).

Example 9: TBkBarrierOmp.C

To compile, run make tbkbomp. This is the OpenMP version of the previous example.

9.8.3 PROGRAMMING A SIMPLE SPMD THREAD POOL

The next example presents another interesting application of the blocking barrier synchronization pattern: running and driving a simple SPMD thread pool. A SPMP (Single Program, Multiple Data) thread pool is a team of worker threads that operate exactly like our well-known SPool utility. In fact, this example exhibits some of the inner workings of the SPool pool. It will certainly contribute to sharpening your understanding of thread synchronization. Here is the global organization of this example:

- First, a set of worker threads is launched. They wait on a blocking barrier, and are next released *after being told what task to execute next*.
- After completing their task assignment, the worker threads are again blocked at the blocking barrier, waiting for their next assignment.
- When the time comes, the main thread releases again the worker threads from the blocking barrier synchronization point, *after informing them of the task to execute next*.

For the sake of clarity, some selected critical parts of the code are discussed in detail.

How the worker threads are launched

Let us first consider how to launch a set of working threads that wait to be told what to do next. Here is the code for the thread function executed by the worker threads when they are created.

```
BkBarrier *BB;            // global blocking barrier
void (*task)();           // a global pointer to a task function
...
void *ThFunction(void *P)
   {
```

```
// Enter an infinite loop.
for(;;)
   {
   BB->Wait();    // Here worker threads sleep
   (*(task))();   // call the function pointed by task
   }
}
```

LISTING 9.4

SimplePool.C (partial listing).

In Listing 9.4:

- BB is a global pointer to a blocking barrier, initialized by main() when the number of worker threads is known.
- task *is a pointer to a task function that has no arguments and no return value*. In due time, main() will initialize this pointer with the address of a specific task function having the same signature.

The thread function enters an infinite loop where it waits at the blocking barrier, and when released it calls the task function referenced by task. I hope readers appreciate how the BkBarrier class.contributes to the simplicity and elegance of this code.

A task is described in this example with a function that takes no argument and returns void. This is totally arbitrary, and any another convenient signature for the task function could have been used. The only function signature that must be respected is the signature of the thread function passed to the basic libraries. But the task function is just a function called by the real thread function once the thread is running, and its signature is our own personal choice. Return values are not needed in general, but the task function could have taken an arbitrary number of arbitrary arguments.

How main() drives the worker thread activity

Every time main() needs the services of the worker threads, it dispatches a new task by a call to the following function.

```
void Dispatch(void (*TSK)())
   {
   BB->WaitForIdle();
   pthread_mutex_lock(&mytask);
   task = TSK;
   pthread_mutex_unlock(&mytask);
   BB->ReleaseThreads();
   }
```

LISTING 9.5

SimplePool.C (partial listing).

This function receives as argument the address of an explicit task function defined somewhere else in the code (remember, in C-C++ the address of a function is just its name). The main thread:

- waits until the worker threads have finished their previous assignment,
- then it re-initializes the task pointer with the new task function address and
- finally it releases the worker threads.

The global mutex mytask guards access to the global shared pointer task. Notice, again, how the BkBarrier interfaces make this code very simple and elegant.

How worker threads terminate

The worker threads are executing an infinite loop. How can they break away and terminate? This is easy: when the main function no longer needs the services of the worker threads, it submits a specific termination task that calls the thread termination library functions—like pthread_exit() in Pthreads or _endthreadex() in Windows. This forces the normal termination of the worker threads, which can then be joined by the master thread.

Full example

The full example is in source file SimplePool.C. It produces exactly the same output as the first example discussed before, but it is programmed in a more flexible way. Two task functions are defined:

- A NormalTask() task function that executes the same code as in the first example: a timed wait to delay the execution, followed by an identification message in stdout. Notice that this task function just describes the work performed by the thread: all the BkBarrier synchronizations have been factored away.
- An ExitTask() task function that forces the worker thread termination by a call to the appropriate thread termination library function.

As far as the main thread is concerned, it dispatches several times the normal task followed by the WaitForIdle() call and then dispatches the exit task to terminate the operation of the worker threads.

Notice that there are two mutexes in this program. One of them guards naturally the task pointer that is accessed by all threads. The other mutex is used to order the output to stdout (the SafeCout utility could have been used instead). If the threads do not have exclusive access to the screen, their identification messages are very often mixed up.

Example 10: SimplePool_P.C, SimliPool_S.C

To compile, run make spool_p or spool_s. The number of threads is read from the command line (the default is 2). The output of this program is the same as the output of the first example.

The programming strategy and idioms deployed in this example are used in the implementation of the SPool thread pool utility.

9.9 ThQueue<T> CLASS

The synchronization utility discussed in this section—a thread-safe queue—is useful when implementing pipelined software architectures connecting producer and consumer threads. A queue is a FIFO (first in, first out) container where data is extracted in the same order in which it is stored. Data is inserted by producer threads and retrieved by consumer threads. A queue can be implemented as an array where data is inserted at its tail and extracted from its head. Objects of this type are also needed in the implementation of other utilities (thread pools and pipelines).

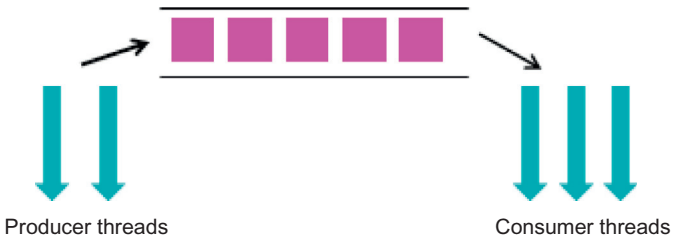

FIGURE 9.5

Thread-safe ThQueue queue.

The C++ STL library has a queue< *T* > template container, where *T* is an arbitrary data type. The STL queue is unbounded, in the sense that if producers are more active than consumers, the queue size can grow indefinitely because data insertions are never blocked. The ThQueue<T> class, sketched in Figure 9.5, extends the STL queue to a more intelligent, thread-safe *bounded* container with a finite capacity where insertions can be blocked, providing more controlled data flow control between producer and consumer threads, as the examples that follow will show.

TBB also proposes bounded and unbounded concurrent queue utilities. The class discussed here is very close to the TBB concurrent_bounded_queue class, with a few extra features providing refined control of the termination of the queue operation.

Design requirements

The design of this utility implements the following features:

- The queue has a finite capacity, fixed when the object is constructed. This finite capacity is useful to provide some degree of load balance by preventing producer threads from running too much ahead of consumer threads.
- Producer threads wanting to insert data in a full queue perform an idle wait until the queue is no longer full (because consumer threads have removed data).
- Consumer threads wanting to extract data from an empty queue perform an idle wait until the queue is no longer empty (because producer threads have inserted data).
- The queue pipeline can be stopped in a clean and efficient way. When some client thread declares the queue as closed, producer threads are no longer able to insert data. Consumer threads are provided by the queue with all the information required to correctly drain a closed pipeline by servicing pending data, and stopping gracefully when no further data is available.

9.9.1 USING THE ThQueue<T> Queue

THE THQUEUE PUBLIC INTERFACE

ThQueue<T> Q(int N)

– Constructor.
– Creates a ThQueue Q of capacity N, holding data items of type T.

`Q.GetSize()`

– Returns the capacity of Q.

`int Q.Add(const T& elem)`

– Adds `elem` to the ThQueue Q. Called by producer threads.
– Returns 0 if the insertion failed because the queue is closed, 1 otherwise.
– If the queue is full, this function does not return. It executes an idle wait until the insertion becomes possible.
– When inserting to an empty queue, this function notifies the *queue not empty* condition that wakes up possible waiting consumer threads.

`T Q.Remove(bool& flag)`

– Returns the next data item in the queue. Called by consumer threads.
– `flag` is an *output parameter*.
– If `flag` is `true`, Q is active and T comes from the queue.
– If `flag` is `false`, Q is closed and T is a fake data item.
– If the queue is empty, this function does not return. It executes an idle wait until the removal becomes possible.
– When removing from a full queue, this function signals the *queue not full* condition that wakes up possible waiting producer threads.

`void Q.CloseQueue()`

– Closes the ThQueue.
– After this call, insertions are not possible and removals behave in the way discussed above, which enables the correct draining of the queue.
– If the queue is full, this function does not return. It executes an idle wait until the insertion becomes possible.

In order to better understand why the member functions behave in the way just described, the issue of correctly stopping a producer-consumer pipeline must be examined. In an application in which some master thread has a global view of the operation and the client code can control whether there is as much data retrieved as data inserted, there is no problem. However, it may also happen that this is not the case, and that the code executed by the consumer threads has no direct knowledge of the producer's activity. In this case, the information that there is no more data coming along the queue must be conveyed by the queue itself.

If there is only one producer and one consumer thread, it is possible to agree on a specific data value to signal the end of the queue operation. The producer thread queues the value, and the consumer thread knows that no more data is available when this value is dequeued. In this case, the `RingBuffer` class introduced in Chapter 8 can directly be used. If, however, there are several producer and/or consumer threads, the issue of terminating the producer-consumer connection is more subtle. To cope with this issue, the `ThQueue` class has internally a flag called `active` that controls the behavior of the `Add()` and `Remove()` member functions. When the queue is created, the `active` flag is always `true`. When the `ClosePool()` function is called by some thread that decides the game is over, the flag is toggled to `false` and, at this point, the producer and consumer threads behave differently:

- **Producer threads**: If they try to insert data after the queue is closed, the `Add()` function does not perform the insertion. It returns immediately with an error code. This solves the problem for producer threads.
- **Consumer threads**: After the queue is closed, there may remain some residual data in the queue that must be dequeued and processed. This is why the `Remove()` function returns a data item, but it

also returns an output Boolean variable flag *that qualifies the accompanying data item returned*, as follows:

- if flag==true, the accompanying data item is valid, because it has been extracted from the queue. The consumer thread knows that it can go ahead with its standard processing.
- if flag==false, the accompanying data item is a fake, invalid data value not taken from the queue because *the queue is closed and empty*. When this happens, the consumer thread knows that it must disregard the returned value and stop removing data from the queue.

In all applications of this utility, client threads must therefore continue to retrieve data items from the queue while checking the Boolean flag returned also by the Remove() function, until they retrieve a data item flagged as false.

9.9.2 COMMENTS ON THE PRODUCER-CONSUMER PARADIGM

The producer-consumer paradigm sustained by this utility is a common parallel construct. It is used internally in other higher level utilities, like the pipeline classes discussed in Chapter 15 or the thread pool NPool utility discussed in Chapter 12. This paradigm, however, must be used with care. Producer threads produce data sets on which consumer threads have to act next. However, if the data sets are important it is totally inefficient to put them in the queue. This forces copying the data set at least two times, for insertion and for removal from the queue, thereby introducing a significant and totally unnecessary memory access overhead, because producer and consumer threads can share global data after all. This is indeed the whole point in multithreading programming. When implementing the producer-consumer paradigm, the queue is in general used to propagate *control information*: indices or pointers to shared data sets, not the data sets themselves.

Queuing data sets forces copying the data because the ThQueue class uses internally a STL queue class, and the STL containers implement *copy semantics*: objects inserted or read from the containers are copied. This is different from the Java containers, which use *reference semantics*: objects are not copied into the containers, which only store a reference to them.

The ThQueue utility is used—in a way transparent to the user—when implementing thread pools. A task corresponds to a C++ object—containing among other things information about the task function to be executed as well as the arguments to be passed to it. Task objects can have arbitrary size, and they are transferred from the thread submitting the parallel job to the thread executing the task. However, only *a pointer to a task object* is queued, with very limited impact on performance.

9.9.3 EXAMPLES: STOPPING THE PRODUCER-CONSUMER OPERATION

Two simple examples follow that demonstrate the mechanism implemented by ThQueue to stop the queue operation. The first one deals with several producers and one consumer, the second one with several consumers and one producer.

Several Producers: TQueue1.C

This example introduces three producer threads and one consumer thread. All producers keep inserting integers until one of them closes the queue. The example shows that the remaining producers stop inserting, and that the consumer thread correctly drains the queue extracting all the residual integers inserted in the queue.

A pool of three producer threads is created. The main thread, rather than waiting in a blocked state, plays the consumer role, extracting integer values and printing them in the screen. The integer values queued depend on the producer threads, so it is easy to know which one of the producers inserted each value printed by main:

- Producer 1 inserts 100 integers, starting from 1. Then, it closes the queue.
- Producer 2 inserts consecutive integers starting from 1000, until the insertion is rejected because the queue is closed.
- Idem for producer 3, with values starting at 100000.

Listing 9.6 shows the implementation of this producer strategy. Running the code, it is verified that producer 1 has inserted the 100 values, and that producers 2 and 3 have at some point stopped inserting. On the other hand, main correctly drains the queue and stops. With the default queue capacity (40 integers) it is observed that producers 2 and 3 have queues substantially with fewer integer values than producer 1, and that the last values inserted change across different runs. This was to be expected, given the limited queue capacity and the fact that producers are blocked when the queue is full. Increasing the queue capacity relaxes the insertion constraints on the producers, and it is observed that the number of items queued by producers 2 and 3 grows.

```
int C;                // queue capacity
SPool  TS(3);         // three producer worker threads
ThQueue<int> *THQ;    // reference to thread safe queue

void ThFct(void *arg)       // Thread function
   {
   int n, start_index, retval;
   int rank = TS.GetRank();

   if(rank==1)    // Producer 1 code
      {
      for(n=1; n<=100; n++) THQ->Add(n);
      THQ->CloseQueue();
      }
   else          // Producers 2 and 3
      {
      if(rank==2) start_index = 1000;
      else start_index = 100000;
      n = 1;
      do
         {
         retval = THQ->Add(start_index+n);
         n++;
         }while(retval);
      }
   }

int main(int argc, char **argv)
   {
```

```
bool read_flag;
int  read_value;

if(argc==2) C = atoi(argv[1]);
else C = 40;
THQ = new ThQueue<int>(C);

TS.Dispatch(ThFct, NULL);    // run threads
// – – – – – – – – – – – – – – – – – – – – – –
// This main thread dequeues and print values
// as long as they are relevant
// – – – – – – – – – – – – – – – – – – – – – –
for(;;)
   {
   read_value = THQ–>Remove(read_flag);
   if(read_flag == false) break;
   else std::cout << read_value << std::endl;
   }
TS.WaitForIdle();   // synchronize with workers
delete THQ;
}
```

LISTING 9.6

Several producers: TQueue1.C.

Several Consumers: TQueue2.C

In the second example, main() acts as producer and queues 200 consecutive integer values. A pool of three consumers is constructed; each consumer dequeues values and prints them to stdout for as long as the value is a valid one. The correct termination is verified. This is the pattern used internally in the thread pool utility discussed in Chapter 12.

Examples 11 and 12: TQueue1.C and TQueue2.C

To compile, run make tqueue1/tqueue2. The queue capacity can be overridden from the command line (the default is 40).

9.10 READER-WRITER LOCKS

Consider an application where several threads are recurrently performing two different operations on a given data set: an operation called generically a **write** operation that modifies the data set and changes its internal state, and another operation called generically a **read** operation that extracts information without modifying the data set or changing its state. An obvious example that comes to mind is a team of threads accessing an ordinary, thread-unsafe, shared STL container. But these generic write and read operations are not necessarily memory operations; they may also be operations in which a shared resource—a shared file, for example—is accessed.

The traditional mutual exclusion mechanism—allowing access of one thread at a time to the data set—may introduce unnecessary synchronization overhead. The write operation must obviously be exclusive, but there is in general no harm in letting several threads perform the read operation concurrently. The read operation only needs the guarantee that the data set is not modified by a write during the read. If the number of threads is important and the reads are much more frequent than the writes, the ordinary mutual exclusion mechanism may induce a severe performance penalty.

Reader-Writer locks—also called shared locks—are optimization utilities designed to safely relax some of the constraints of the strict mutual exclusion mechanism, thereby allowing several threads to perform read operations concurrently. This problem is not as trivial as it may look at first sight. Just locking a mutex for writes but not for reads is not sufficient, because *a write operation must exclude any other write or read*. Moreover, in the particular case of memory operations, memory consistency, as discussed in Chapter 7, enforces a mutex lock for the reads.

Reader-Writer locks are proposed by all programming environments, except OpenMP and C++11. Figure 9.6 shows their operation. Exclusive write sections (black threads) are well separated from concurrent read sections (gray threads). A point that requires careful discussion is the way the transitions from exclusive to concurrent regions (and viceversa) are managed. Indeed, reader-writer locks are particularly efficient in the presence of lots of readers and a few writers. However, if writers are forced to wait for as long as there are ongoing readers, they can eventually starve (never get the lock) in the presence of too many readers. The way of guaranteeing the absence of starvation is discussed later on.

9.10.1 PTHREADS RWMUTEX

In Pthreads, a reader/writer locks works like an ordinary mutex, the only difference being that there are two different functions for locking in read or in write mode. The unlock function is the same in both cases. The programming interface is as follows:

- pthread_rwmutex_t lock declares a reader/writer lock.
- Its address is passed to pthread_rwmutex_init() for initialization. There are no attributes for this type of mutex. Destruction works in the same way.
- pthread_rwmutex_rdlock(&lock) locks lock in read mode.

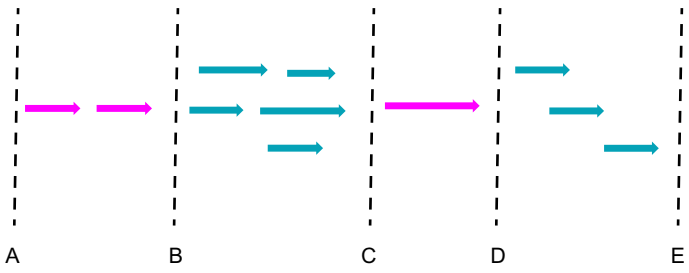

FIGURE 9.6

Operation of a reader-writer lock. Black threads are writers, and gray threads are readers.

- pthread_rwmutex_wrlock(&lock) locks lock in write mode.
- pthread_rwmutex_unlock(&lock) unlocks lock in any mode.

The man pages provide additional information (look, for example, at man pthread_rwlock_init).

9.10.2 WINDOWS SLIM READER-WRITER LOCKS

Reader-writer locks in Windows were introduced by Windows Vista. The programming interface is described below. As for Pthreads, I have not found in the documentation any reference to the ordering policy for write-read transitions. An educated guess is that starvation is guaranteed not to occur, using policies of the kind described later on.

Windows introduces a new synchronization object, SWRLOCK, that must be declared with global scope, and its address passed to the initialization, acquire, and release functions. Here is a summary of the programming interface, taken from the Windows API online documentation [13]:

SWRlock member functions

- SWRLOCK lock declares a SWRlock.
- InitializeSWRLock(&lock) initializes the reader-writer lock.
- Locking in shared mode. Called by read operations.
  - void AcquireSWRLockShared(&lock).
  - int TryAcquireSWRLockShared(&lock). Does not wait. Returns 0 if mutex is unlocked, or 1 if mutex is locked.
  - void ReleaseSWRLockShared(&lock).
- Locking in exclusive mode. Called by write operations.
  - void AcquireSWRLockExclusive(&lock).
  - int TryAcquireSWRLockExclusive(&lock). Does not wait. Returns 0 if mutex is unlocked, or 1 if mutex is locked.
  - void ReleaseSWRLockExlusive(&lock).

Notice that the TryAcquire....() functions listed above have only been supported since Windows 7.

9.10.3 TBB READER-WRITER LOCKS

TBB introduces two reader-writer mutex classes, spin_rw_mutex and queuing_rw_mutex. They have exactly the same design profiles as the standard spin_mutex an queuing_mutex classes discussed in Chapter 5, and they operate in the same way, with a few minor modifications:

- These reader-writer mutexes are always locked using the internal scoped_lock classes.
- TBB introduces an extra Boolean parameter called write that is passed to the scoped_lock functions. If writer is true(false) the mutex is locked in exclusive(shared) mode.
- There are two additional member functions allowing the programmer to upgrade a reader mutex to writer or downgrade a writer mutex to reader, but we do not have simple examples to propose to illustrate their relevance.

Listing 9.7 shows two ways of locking a spin_rw_mutex in exclusive (writer) mode. To lock in shared (reader) mode, change the write parameter to false.

```
using namespace tbb;

spin_rw_mutex   RWmutex;   //  create RWmutex

// Explicit locking:
spin_rw_mutex::scoped_lock  Slock;       // construct scoped lock
Slock.acquire(RWMutex, bool write=true); // lock RWmutex
...
SLock.release();                         // unlock RWmutex

// Scoped locking:
   {
   spin_rw_mutex::scoped_lock  MLock(RWmutex, bool write=true);
                                              // RWmutex is locked
   ...
   }
```

LISTING 9.7

Locking TBB reader-writer mutexes.

The TBB documentation indicates that the reader/writer locks have write priority. This means the reader-writer lock is fair and read requests are blocked when there is a pending write request, as will be discussed in the next section.

9.11 RWlock CLASS

The RWLock class is a simple wrapper of the native Pthreads and Windows reader-writer utilities. The member functions are listed below. *This class does not have a C++11 implementation*, because there are no native reader-writer lock services in C++11. One option in this environment is to use the TBB or Boost shared mutex classes.

RWLOCK INTERFACES

RWLock()

- RWLock constructor.
- Creates a reader-writer lock.

int Lock(bool mode)

- When mode is true-false, locks in write-read mode. .
- Returns error code in Pthreads.
- Return value is not relevant in Windows.

int TryLock(bool mode)

- When mode is true-false, tries to lock in write-read mode.
- This function never waits.

- Returns 1 if lock is acquired, 0 otherwise.

int Unlock(bool mode)

- Unlocks a reader-writer lock in the mode it was locked.
- Returns an error code in Pthreads.
- Return value is not relevant in Windows.

9.11.1 EXAMPLE: EMULATING A DATABASE SEARCH

A first example, well adapted to the functionality of a reader-writer lock, is the database search example developed in Section 3.10. In Chapter 3, a built-in feature of the SPool utility, enabling the cancellation of a parallel job, was used. Here, the cancellation of the parallel region is implemented by reading and writing a shared flag. The worker threads keep reading the flag—initially false, called interrupt in Listing 9.8—to check if cancellation is requested. The thread that requests the cancellation toggles the flag to true. There are therefore a huge number of reads, and only one write: an ideal use case for testing the reader-writer lock performance.

The RWLock class is used in this example. The partial listing of the source file DbSearch_P.C that follows is very close to Listing 3.30 in Chapter 3. The main() function is the same. In the thread function the check for cancellation is performed by reading the flag with the lock in read mode, and the write to cancel the process is made with the lock in write mode.

```
SPool *TS;
const double EPS = 0.000000001;
const double target = 0.58248921;
Data D;

bool   interrupt;   // NEW, initially false
RWLock rw_lock;     // NEW, protects flag

// The workers thread function
// - - - - - - - - - - - - - - -
void th_fct(void *arg)
    {
    double d;
    int rank = TS->GetRank();
    Rand R(999*rank);

    for(;;)
       {
       // - - - - - - - - - - - - - - - - - - - - - - - -
       // check the interrupt flag and eventually break
       // - - - - - - - - - - - - - - - - - - - - - - - -
       rw_lock.Lock(false);
       my_flag = interrupt;
       rw_lock.Unlock(false);
```

Continued

```
        if(my_flag) break;
        // - - - - - - - - - - - - - - - - - - - - - - - -

        d = R.draw();
        if(fabs(d-target)<EPS)
           {
           D.d = d;
           D.rank = rank;
           // - - - - - - - - - - - - - - - - - -
           // Request interruption and break
           // - - - - - - - - - - - - - - - - - -
           rw_lock.Lock(true);
           interrupt = true;
           rw_lock.Unlock(true);
           break;
           // - - - - - - - - - - - - - - - -
           }
        }
     }
```

LISTING 9.8

DbSearch_P.C: simulating a database search. The windows version DbSearch_W.C is similar.

Example 11: DbSearch_PW.C

To compile, run make dbs_pw. The number of threads is 4 by default. It can be overridden from the command line.

The observed performance is not as good as in the Chapter 3 example using the SPool built-in job cancellation service, based on the native Pthreads thread cancellation utilities. But it is better than the standard mutex-locking performances.

9.11.2 EXAMPLE: ACCESSING A SHARED CONTAINER

An example is provided in the file VAccess.C. The target data set whose thread safety is enforced is an STL integer vector container std::vector<int>. This container is repeatedly accessed by a set of reader and writer threads. The initialization code reads the number nTh of threads from a data file, *and creates two different thread pools of* nTh *threads each*: one for reader and the other for writer threads. The code organization is described below using the RWLock interfaces.

Writer threads

- The integer values each writer thread inserts in the vector container depend on its rank. A thread of rank k starts inserting at the integer value v = 10 k, and continues from there on.
- Each writer thread performs the following steps five times:
  - It waits for a random time interval in the range [0, 2000] milliseconds.
  - It increments v and calls Lock(true).
  - It inserts v at the end of the container, and calls Unlock(true).

- After operating five times, the thread terminates.

The write operations occur randomly, but the container values are related to the thread ranks as described above. When looking at the container content, one knows from which thread each inserted item comes from.

Reader threads

The reader threads repeatedly access the container to read and print its content to stdout. They execute a read loop until the main thread signals the end of the operation. At each iteration of the read loop:

- Reader threads wait for 250 ms, in order to slow down the read operations.
- They call Lock(false), and print the container to stdout, using an utility function provided by the vath library.
- Finally, they call Unlock(false) to close the read operation.

The main thread interrupts the read operations after a time interval (in milliseconds) read from the data file. Once the read operation is interrupted, the main thread joins the reader and the writer teams.

Example 12: VAccess.C

To compile, run make vaccs. The number of writer and reader threads, as well as the duration of the reads, are read from the data file vaccs.dat.

The code output shows several times the content of the container on stdout (as many times as read operations took place), and also shows how the container grows as interleaved write operations take place.

This example shows how having two (or more) independent thread pools in the same application may simplify the programming of some parallel contexts. A more elaborate example is developed in Chapter 12.

A TBB version.

The previous code, using RWLockVA, is portable. A TBB version is proposed in VAccessTbb.C, in order to provide an example of the usage of the TBB shared mutex class.

Example 12: VAccessTbb.C

To compile, run make vaccstbb. The number of writer and reader threads, as well as the duration of the reads, are read from the data file vaccs.dat.

9.11.3 COMMENT ON MEMORY OPERATIONS

The previous example involves *memory operations*, and it seems to contradict the conclusions of Chapter 7 concerning memory coherency: exclusive writes are naturally protected by a mutex, but read operations must lock the same mutex to make sure they capture the last written value to the memory location involved. Looking back to Figure 9.6, it is clear that read operations are not locking anything:

otherwise, they could not act concurrently. Therefore, we may ask: *how do we know that memory consistency is guaranteed for memory operations?*

The answer is that the write and the read sectors in Figure 9.6 are separated by memory fences located at boundary points A, B, C, D, and E, provided by an internal mutex. This guarantees that memory will be coherent every time the read operations start.

9.12 GENERAL OVERVIEW OF THREAD POOLS

It should be clear by now that a parallel job is composed of a collection of logical, weakly coupled units of sequential computation: the tasks. Parallelism is implemented by mapping tasks to threads for parallel execution, with occasional synchronizations required by the algorithm integrity. All the examples and applications discussed until now exhibit a simple fork-join parallel context in which the N tasks of a parallel job are uniquely mapped to a set of N distinct worker threads. This is indeed the basic operational model of the simple SPool utility and the OpenMP parallel directive that has been used all along.

More complex applications may require more than a simple fork-join parallel pattern. An application may need to repeatedly activate a varying number of worker threads in order to perform a substantial number of different parallel tasks. A network server may need to request the services of a new thread to treat the connection established by a new client. Rather than creating and destroying the worker threads as they are needed, it is much more efficient to dispose of a stock of blocked threads sleeping in an idle wait state, and wake them up to execute new tasks when needed. An example was given early in this chapter, using a blocking barrier to manage the worker threads. Indeed, waking up sleeping threads is more efficient than creating them from scratch every time they are needed (which involves, among other things, creating a new stack buffer for each new thread). It may also happen that the number of tasks to be executed by a parallel job is not known in advance. This is indeed the case of recursive *divide and conquer* algorithms, where tasks recurrently split in two or more smaller tasks as long as the computational workloads are bigger than some predefined granularity.

This leads to the introduction of *thread pool* environments to implement efficient task management. A thread pool is a collection of worker threads, by default sitting silently in the background executing an idle wait, waiting to be woken up to perform a task. Worker threads go back to an idle wait when no new tasks requests are available. Their presence in an application does not reduce the available CPU resources when the pool is not active.

A parallel treatment is a collection of tasks, not threads. The precise implementation of the task concept depends of course on the particular thread pool programming environment, but in all cases it incorporates precise information about the function that describes the computation to be performed. In the simple thread pools involved in the SPool utility of the OpenMP parallel region, there is a one-to-one mapping of tasks to threads, and tasks start running as soon as the parallel treatment is launched. However, we will see in the following chapters this is not the most general case, and programming environments are designed to cope with contexts in which the number of tasks submitted for execution is bigger than the number of worker threads in the pool. As tasks are submitted, they are queued in some way until the worker threads finish or suspend previous assignments and become available to service new tasks.

The way tasks are mapped to threads depends on the particular programming environment. In some cases, task scheduling is fair and tasks are served on a first come, first served basis. In other cases, sophisticated unfair strategies may be deployed to decide which task is executed next. This is the case in OpenMP and TBB.

Three different thread pools are considered in the following chapters:

- **A—OpenMP**, a well-established, widely adopted programming environment, available in FORTRAN and C-C++. OpenMP played a leading role in enabling the adoption of shared memory programming in SPMD platforms. A complete coverage of OpenMP, including the most recent 4.0 release, is presented in Chapter 10.
- **B—TBB** A C++ library proposing a substantial number of standalone utilities for multithreading, and a task-centric thread pool programming environment using sophisticated strategies to optimize the mapping of tasks to threads. TBB is discussed in Chapters 11 and 16.
- **C.—NPool class**, a C++ utility for the management of a team of worker threads, implementing many of the OpenMP and TBB features. The scheduling strategies for managing parallel tasks are less ambitious than in these environments. Nevertheless, the interest of this utility is that it implements a slightly different programming style that simplifies the deployment of some specific parallel contexts. This utility is discussed in Chapter 12.

Clearly, different design choices induce some differences in user interfaces and programming styles. Nevertheless, a substantial number of pitfalls and best practices are common to all of them. Insight into multithreading is considerably enhanced by comparing the different ways in which they are resolved, and by keeping in mind the different development strategies they enable.