CHAPTER 6

EVENT SYNCHRONIZATION

6.1 IDLE WAITS VERSUS SPIN WAITS

It often happens in concurrent or parallel programming that a given thread reaches a point in which nothing useful can be done before other threads have completed some part of their workload providing the information required to pursue the ongoing task. The fact that expected information has become available is considered as an *event*. The initial thread needs eventually to wait for the occurrence of this event before pursuing its execution stream. The event synchronization mechanisms discussed next allows threads to wait for expected events.

When discussing mutexes, it was observed that a thread waiting to lock a mutex could wait for the mutex availability in two ways: moving to a blocked state and liberating the executing CPU (*idle wait*), or keeping ownership of the CPU by executing a do-nothing instruction while waiting (*busy wait*). This is a particular case of event synchronization, the event being in this case the mutex availability. These two options apply to *any* kind of event synchronization, and we proceed to describe them in a more precise way.

The synchronization mechanisms discussed here are universal in the sense that, barring minor semantic differences, they operate the same way in all basic libraries: Pthreads, Windows, and the C++11 standard. We start by proposing a precise definition of an event:

An event is described by any boolean expression, called *predicate*, that switches values passing from *false* to `true` or vice-versa. The event occurs when the predicate is toggled.

Idle wait, also called system wait

The idle wait mechanism *is a native library synchronization primitive*, operating in the following way:

- When thread A needs to wait for the value `true` of a global predicate, it calls a function that puts A in a blocked state if `predicate==false`, waiting for the expected event to happen. The function blocks, and returns only when the wait is terminated and A is rescheduled.
- When the time comes, another thread B toggles the predicate to `predicate=true` and wakes up thread A in a way to be described next.
- This mechanism is robust. While waiting, thread A yields the executing core, which becomes available to run other threads. Thread A is not rescheduled to run until it wakes up. When this happens, A is moved to the ready pool, and is rescheduled when a core becomes available.

- This protocol requires the participation of a communication agent triggered by thread B to wake up thread A. Such a communication agent is called a *condition variable*.

Busy wait, also called spin wait

The busy wait *is not* a native synchronization primitive. It must be explicitly programmed, as it will be shown later on. It operates in the following way:

- Thread A, that needs to wait for an event, executes a do-nothing loop of the kind `while(!predicate)`, for as long as the predicate is false.
- This avoids system calls, but the waiting thread remains scheduled using CPU resources for the whole duration of the wait.
- This looks simple and straightforward, but the mechanism is delicate, because the waiting thread keeps reading a predicate value in memory waiting for an update coming from another thread, and the fundamental memory consistency issue—*when does a memory value written by a thread become visible to other threads?*—cannot be avoided.
- The do-nothing loop does not work as such. Memory consistency requires further synchronizations in order to have a spin wait working correctly. Without them, thread A may never seen the update made by another thread B. Chapter 7 discusses how to program a spin wait, after exploring in more detail the memory consistency issue.
- Notice that, since the waiting thread remains active, no wake up notification agent is needed. Threads communication operates directly through the monitoring of a shared predicate.

Plan for the rest of the chapter

First, a careful discussion of how the first mechanism operates in Pthreads, Windows, and C++11 is presented. OpenMP, which focuses on high-level, easy-to-use synchronization constructs, does not provide a programming interface for idle waits. TBB did not initially propose this basic synchronization primitive either because, as discussed in detail later on, its direct usage is unsafe in the TBB thread management environment. But recently TBB has incorporated an implementation of C++11 synchronization as a standalone utility that naturally includes the idle wait synchronization primitive.

The idle wait discussion that follows focuses on understanding *why this synchronization primitive is designed in this way*. But this basic idle wait mechanism will rarely be used directly. It will, most of the time, be encapsulated in high-level, easy-to-use objects implementing complex synchronization patterns, like barriers, thread pools, pipelines, etc., allowing the programmer to focus on *what to do*, not *how to do it*.

Nevertheless, going through this discussion is in my opinion a good pedagogical exercise that enhances the understanding of concurrent programming. The programming idioms discussed in this section, as well as the specific examples that follow, are rather simple, and they are the same in all libraries and programming environments. They will first be introduced using Pthreads, and then their Windows and C++11 implementations will follow.

At the end of this chapter, some useful utilities introduced by C++11 to implement event synchronization—*futures* and *promises*—are discussed. They have a more restricted scope than condition variables, but they are portable and easy to use in application programming.

6.2 CONDITION VARIABLES IN IDLE WAITS

Condition variables are, like mutexes, opaque data structures in Pthreads—their type is pthread_cond_t—or Windows—their type is CONDITION_VARIABLE. In C++11, they are object instances of the std::condition_variable class. Pthreads also introduces (very rarely used) condition variables attributes, of type pthread_condattr_t, which are initialized by calling functions that specify their properties, exactly as was the case for threads and mutexes. We will never need in this book to go beyond condition variables with default attributes, whose initialization is very simple. In Windows and C++11, condition variables attributes are not needed.

The role of condition variables is simply to establish a link between cooperating threads that need to communicate on the basis of an event. Condition variables behave like wake-up clocks: they ring when an event happens under the action of a thread that signals an event, but they know nothing about the *nature* of the event or the threads they are waking up.

Condition variables are agents that signal that something has happened. They are associated to a contract between cooperating threads, but they know nothing about the nature of the contract or the threads to which their signal is addressed.

- When a thread decides to wait, it calls a library function to which it passes a condition variable. This means: *I am getting out of the way until this condition rings and wakes me up*. The library function clearly does not return until the calling thread is active again.
- When a thread decides to wake up sleeping threads, it calls a library function to which it passes a condition variable. This means: *wake up one—or all—the threads currently waiting on this condition*.
- That is all there is. The precise event that is triggering these actions does not directly show up in these transactions.

The fact that a condition variable has no direct knowledge of the nature of the expected event may seem at first sight somewhat strange. Obviously, condition variables and event predicates are related to each other, but *maintaining this relationship is the programmer's responsibility*. This feature adds in fact a substantial amount of flexibility to this event synchronization primitive, as the examples at the end of the chapter will show.

Three data elements are therefore intimately related in any event synchronization mechanism:

- A shared data item, the predicate, whose change of value determines the occurrence of the synchronizing event.
- A mutex that guards the access to the boolean predicate.
- A condition variable used for signaling the synchronizing event.

It is good programming practice to declare these variables together, whenever possible. Before getting involved with the basic libraries condition variables programming interfaces, let us take a look at some declarations of global synchronization variables.

```
bool  predicate;

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;    // Pthreads
pthread_cond_t  mycond  = PTHREAD_COND_INITIALIZER;
```

Continued

```
CRITICAL_SECTION    CS;               // Windows
CONDITION_VARIABLE  CV;

std::mutex                mymutex;   // C++11
std::condition_variable   mycond;
```

LISTING 6.1

Declaring event synchronization objects.

In all cases, a mutex that protects a predicate is declared, as well as a condition variable related to the predicate. In Pthreads, a static initialization is used, that adopts the default attributes for these variables. This will be largely sufficient for our purposes in this book. In Windows, the condition variable must be initialized with a function call given below. The C++11 declarations do not require further initializations, because they just call the no argument constructor of the corresponding classes. Notice also that the predicate related to the expected event does not need to be an explicit boolean variable. It can also be a boolean expression (namely, an expression that evaluates to true of false) involving arbitrary data types.

6.3 IDLE WAITS IN PTHREADS

This section deals with the idle wait implementation in Phreads: first the programming interfaces, and then the details of the wait protocol. *Keep in mind that the wait protocol discussed in a Pthreads context operates in exactly the same way in all other basic libraries.* After considering Pthreads in detail, the implementation of these same concepts in Windows and C++11 is discussed.

PTHREADS CONDITION VARIABLE INTERFACE

int pthread_cond_destroy(*cond)

– Destroys the condition variable whose address is passed as argument.

int pthread_cond_wait (*cond, *mutex)

– Puts the thread to wait on the condition passed as argument.
– The wait is unlimited. If the condition is not signaled, the thread waits forever.
– This function blocks until the waiting thread wakes up and is rescheduled. The mutex whose address is passed is the mutex that guards the predicate.

int pthread_cond_wait(*cond, *mutex, *date)

– Puts the thread on a *timed wait* on the condition passed as argument.
– If the condition is not signaled before the date passed as third argument, the wait is timed out, thread wakes up, and this function returns 1.
– Otherwise, the thread wakes up when the condition variable is signaled, and the function returns 0.
– struct date is a particular date data item defined in sts/times.h.

int pthread_cond_signal(*cond)

– Signals the condition variable passed as argument.
– Wakes up *one* of the threads waiting on the condition.

int pthread_cond_broadcast(*cond)

– Signals the condition variable passed as argument.
– Wakes up *all* the threads waiting on the condition.

6.3.1 WAITING ON A CONDITION VARIABLE

There are two functions a thread may call to wait on a condition variable. The first one is the simplest, in which the thread waits forever until the event is signaled. The second one performs a *timed wait*: by passing a time interval: the thread goes to sleep for a number of milliseconds, and wakes up if it has not been waken up before by the signaling of the condition variable. This timed wait for an event—a feature in all basic libraries—is a very powerful service. It is seating under the hood of useful high-level utilities introduced in Chapter 9.

It may be surprising to see *the mutex that protects the predicate involved in the wait on condition call*. This is not obvious, and it is in fact related to a subtle issue in the wait protocol discussed next.

6.3.2 WAIT PROTOCOL

How is a thread put to wait on a condition? Imagine that the predicate variable is an integer count, and that the purpose is to synchronize threads on the condition count==0. This is the protocol *that must be used in all cases*.

```
int          count;
pthread_mutex_t r_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t     cond = PTHREAD_COND_INITIALIZER;

void *thread_function(void *s)
   {
   ...
   pthread_mutex_lock(&r_mutex);
   while(count != 0) pthread_cond_wait(&cond, &r_mutex);
   pthread_mutex_unlock(&r_mutex);
   ...
   }
```

LISTING 6.2

Wait protocol

There are several relevant issues in this apparently innocent piece of code. The top of Listing 6.2 shows the default static initialization of the mutex and the condition variable linked to the predicate count. Furthermore, notice that, in the body of the function:

- The mutex is first locked because the predicate count must be accessed for testing its value.
- The code enters a while() loop and calls pthread_cond_wait(), if the predicate is not true. *This function does not returns immediately*. It enters a wait *with a locked mutex*, and returns—again *with a locked mutex*—when the thread is rescheduled after wake up.
- Because the mutex is always locked on return, the code can execute a new iteration of the while loop and check again the predicate. If it is true, the thread breaks away from the while loop and unlocks the mutex. Otherwise, the thread calls again pthread_cond_wait() and starts a new wait.

Now, one may wonder how is it possible, if the mutex has been locked all the time, that some other thread had toggled the predicate? Figure 6.1 describes what happens *inside* the pthread_cond_wait() function:

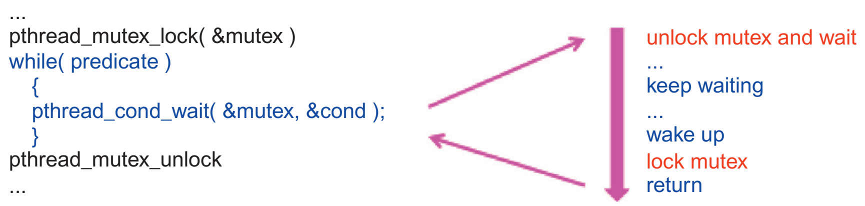

FIGURE 6.1

Wait on condition mechanism.

- This function starts by *atomically* unlocking the mutex passed as argument and setting the thread in a wait state. Then it waits peacefully until the condition is notified.
- The mutex is therefore unlocked while the thread is waiting. Otherwise, nobody else would ever be able to access and modify the predicate. However, *before returning, the wait function locks again the mutex passed as argument*.
- Now we can understand why the mutex must be passed to the function that executes the wait on condition: she is in charge of unlocking the mutex during the wait, and locking it again after the thread wakes up.

Why should the predicate be checked again on return?

Another subtle point to be observed is that this protocol is tailored to force a new check of the predicate after return from the wait. A `while{}` loop is mandatory for this to happen; using a conditional `if()` statement will not force the check of the predicate after return from the wait.

This check is imposed by thread safety, to prevent a subtle race condition from happening. Indeed, the above described wait protocol leaves a tiny window open for a race condition. It is not impossible that, between the condition variable notification that wakes up the sleeping thread and the mutex re-lock, another thread comes in and changes again the predicate. If this is the case, when the function returns and the predicate is checked, the thread goes back to a wait as it should. Therefore, the `while()` loop is mandatory for thread safety.

Why an atomic mutex lock and wait?

The just described wait protocol raises another interesting question. Why are the lock-unlock mutex actions delegated to the `pthread_cond_wait()` function itself? The reason is that the two actions—unlocking the mutex and moving the thread to a blocked state—must be done atomically, which means this compound operation must look instantaneous to other threads. Otherwise, there is the risk of missing a condition variable notification. Indeed, consider the following scenario:

- The active thread that, finding the predicate not true, has decided to wait, is preempted *right after unlocking the mutex, but before it had time to register as a waiting thread.*
- One could argue that there is, in principle, no problem. This thread—henceforth called thread A—should be able to register and go to wait later on, when it is next scheduled.
- However, it may happen that other threads that are scheduled in the meantime have set the predicate to `true`. If this happens, one of them has notified the condition on which thread A should be waiting.

- But thread A *is not waiting*, precisely because it has not yet registered as a waiting thread. Therefore, thread A misses this notification and, when it is scheduled again, it waits on a condition that has already been notified, *with the wrong value of the predicate*.

This kind of behavior is obviously incorrect, and the only way to prevent it is *to have an atomic mutex unlock and wait*: these two actions are tied together in such a way that threads just cannot not be preempted in the way described above. This is exactly what is guaranteed by the wait on condition function calls. The need to cope with this problem forces the scheme adopted by all the basic libraries (Pthreads, Windows, C++11).

6.3.3 WAKING UP WAITING THREADS

The Pthreads library uses two notification mechanisms for waking up waiting threads: one is called *signal* and the other is called *broadcast*. The reason is that there may be more than one thread waiting on a given condition variable cond.

A SIGNAL wakes up only ONE thread waiting on the condition variable. A BROADCAST wakes up ALL threads waiting on the condition variable.

The pthread_cond_signal() function wakes up only one thread waiting on the condition cond. Which one is actually waken up is not controlled by the programmer, it is decided by the scheduler. The function pthread_cond_broadcast() wakes up all threads waiting on the condition cond.

Which one of this two functions is required depends then on the context. In a barrier synchronization, for example, threads go to sleep as they reach the barrier synchronization point, and the last thread that reaches the barrier wakes up all other threads. In this case, a broadcast is mandatory.

6.4 WINDOWS CONDITION VARIABLES

The initial versions of the Windows operating system did not provide support for condition variables. Windows has a global strategy for waiting for things to happen: threads to terminate, mutexes to be unlocked, events to occur. In all cases the relevant object is created, a HANDLE is returned and used by client code to query the target object or to request services from it. The Windows kernel objects, like the condition variables met before, signal events but their programming interfaces do not follow the condition variable wait protocol presented before, because the associated wait functions are not tailored to receive a predicate guarding mutex as argument.

Fortunately, Windows Vista introduced condition variables in the Windows API, which operate in exactly the same way as the condition variables in the other basic libraries, making code migration a very simple affair. Here is the Windows condition variable programming interface. Full details are available in the Window API online documentation [13].

WINDOWS CONDITION VARIABLE INTERFACE

CONDITION_VARIABLE CV

– Declares a CONDITION_VARIABLE object, called CV (the name is arbitrary).

– This object must be associated to a CRITICAL_SECTION object that guards the predicate.

void InitializeCoditionVariable(&CV)

– Initializes the CONDITION_VARIABLE object CV.

bool SleepConditionvariable(&CS, &CV, DWORD msecs)

– Sleeps on the condition variable CV.
– This is in general a timed wait, for a duration of msecs milliseconds.
– For an unlimited wait, the symbolic constant INFINITE must be passed as third argument.
– Return value is true if thread has been effectively waken up by the condition variable.
– Return value is false if wait interval has expired, or other error occurred.
– See discussion below for the way to handle a false return.

void WakeConditionVariable (&CV)

– Wakes up *one* thread waiting on the condition variable.

void WakeAllConditionVariable (&CV)

– Wakes up *all* threads waiting on the condition variable.

Handling timed waits.

When the return value of the Sleep ConditionVariableCS() function is false, it must be decided whether the wait has been timed out or whether another error has occurred. This is achieved by calling the function GetLastError(DWORD *error) that returns in the output parameter error an error code. If the error code returned equals the symbolic constant ERROR_TIMEOUT, we have a normal return resulting from a timed out wait. This approach is used in our Windows implementation of the BLock utility to be discussed in Chapter 9.

Why does the name in the Sleep...() function ends with a "CS"? The reason is that this function requires a condition variable associated with a critical section object to guard the predicate. There is another wait function call SleepConditionVariableSWR(), in which the condition variable is associated with a *slim read-write lock*, also introduced by Windows Vista, which will be discussed in Chapter 9. Again, this feature is introduced to optimize certain parallel contexts, and it will never be used it in this book.

Listing 6.3 summarizes the Window condition variable wait protocol for an unlimited wait. As you can appreciate, it is identical to the Pthreads protocol in Listing 6.2.

```
bool predicate;
CRITICAL_SECTION CD;
CONDITION_VARIABLE CV;

void *thread_function(void *s)
   {
   ...
   EnterCriticalSection(&CS);
   while(!predicate)
      SleepConditionVariableCS(&CS, &CV, INFINITE);
   LeaveCriticalSection(&CS);
   ...
   }
```

LISTING 6.3

Windows wait protocol

6.5 C++11 CONDITION_VARIABLE CLASS

Idle waits for an event are also implemented in C++11 using condition variables. The C++11 standard defines two condition variable classes as well as an enumeration class that defines return values for timed waits.

```
namespace std
   {
   enum class cv_status{timeout, no_timeout};

   class condition_variable;
   class condition_varyable_any;
   }
```

LISTING 6.4

C++11 condition variable classes

Consider first the std::condition_variable class. Besides a simple no argument constructor, this class provides the member functions listed below, which operate in the same way as the previous C interfaces in Pthreads and Windows. Naturally, there is a slight difference in the fact that these C++ member functions are called on a std::condition_variable object, whose address is implicitly passed by the compiler via the this pointer.

C++11 CONDITION VARIABLE INTERFACE

void notify_one()

– Wakes up *one* thread waiting on the caller condition variable object.

void notify_all()

– Wakes up *all* threads waiting on the caller condition variable object.

void wait(std::unique_lock<std:: mutex>& lock)

– Performs an unconditional wait on the caller condition variable object.
– Using a std::unique_lock<std::mutex> for locking in the code block in which the wait is performed is mandatory. See the comments below.

template< typename R, typename P>
cv_status wait_for(unique_lock<mutex>& lock, **const std::chrono::duration<R, P> time_interval**)

– The two template parameters are the types needed to define a duration object, to represent a time interval.
– Performs a timed wait on the caller condition variable object.
– If the condition is not notified during the elapsed time interval passed as second argument, this function returns timeout.
– If, instead, the condition is notified the function returns no_timeout.
– See the comments below, and the duration discussion in the Annex of Chapter 3.

There is another version of the timed wait, wait_until(), which involves an absolute time date for the end of the wait, instead of a time duration. It is not listed above because it will never be used in this book. The usage of duration data items has already been discussed in Chapter 3, when examining the way a thread puts itself to wait for a predefined time interval. Examples of timed waits are given below.

The wait() functions described above implement the same protocol previously discussed in Pthreads and Windows. It is easy to understand why a std::unique_lock locking a std::mutex is needed in the C++11 implementation. As we know, the wait() function needs to internally unlock the mutex during the wait and lock it again it before the return, and this is not possible with a lock_guard scoped lock. The extended flexibility of the unique_lock is required in this function call.

The other class, std::condition_variable_any, works with any kind of scoped locks and mutexes. This class has internally a more sophisticated implementation. As far as we are concerned, the features provided by the simpler std::condition_variable class are largely sufficient for our purposes.

Listing 6.5 summarizes the C++11 condition variable wait protocol for an unlimited wait. It is of course identical to the Pthreads and Windows protocols discussed before.

```
bool predicate;
std::mutex my_mutex
std::condition_variable cv;

void *thread_function(void *s)
   {
   ...
      {
      std::unique_lock<std::mutex> my_lock(my_mutex);
      while(!predicate)
        cv.wait(my_lock);
      }
   ...
   }
```

LISTING 6.5

C++11 unlimited wait protocol

NOTE: a portable utility is proposed in Chapter 9, encapsulating in a easy-to-use programming interface the Pthreads, Windows, and C++11 event synchronization protocols with unconditional or timed waits.

Another form of the wait() functions

C++11 proposes another version of the idle wait functions, which incorporates the predicate in the function call and avoids the need for using the while() loop to check for spurious wakeups. In this case, an additional argument must be passed to the wait() functions: a pointer to a function—or a function object—that takes no arguments and returns the value of the predicate. Here are the function declarations, taken from the C++11 documentation [14]. They will be clarified with an explicit example.

```
template< typename Pred>
void wait(unique_lock<mutex>& lock, Pred pred)
```

- Performs an unconditional wait on the caller condition variable object.
- This function performs internally the while loop that checks again the predicate.
- The function only returns if the predicate is true.

template< typename R, typename P, typename Pred>
bool wait_for(unique_lock<mutex>& lock, **const std::chrono::duration<R, P> time_interval, Pred pr**)

- Performs a timed wait on the caller condition variable object.
- Returns false if the condition is not notified during the elapsed time interval passed as second argument.
- Returns true if the condition is notified and the predicate is true.

6.6 EXAMPLES OF IDLE WAIT

We propose next a couple of examples that exhibit the way the idle wait primitive works in practice. Since these codes are not portable because they target explicitly the basic libraries, there are separate Pthreads, Windows, and C++11 versions of each example. The Pthreads version is used below to exhibit the way the codes are organized, but the Windows and C++11 versions are identical, barring the mild semantic differences just discussed.

6.6.1 SYNCHRONIZING WITH AN IO THREAD

Let us assume that the main thread dispatches a worker thread to perform a lengthy I/O operation, and at some point later on it needs to wait for its completion. There is a trivial way of handling this problem: the main thread just joins the worker I/O thread. However, it may happen that the worker thread is not joinable, or that it must stay alive to continue to do something else after completing the I/O operation. In this case, the way to proceed is to put the main thread to wait on a condition, and let the worker thread signal the condition when the I/O operation is completed.

To simulate a lengthy I/O operation, this example uses a utility to be discussed in Chapter 9: a Timer object. Timer objects are local to a thread. When the owner thread calls its Wait() member function, the thread goes to sleep for the number of milliseconds passed as argument. We have seen before that this is easily done in C++11 and Windows. The Pthreads implementation, however, involves some sophisticated programming. The Timer utility encapsulates the three implementations in a common interface. A Timer will be used to stop the worker thread for 2 s.

The Pthreads source code for this example is in file IoTask_P.C. Here is the listing.

```
bool  flag;
pthread_mutex_t flock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t fcond  = PTHREAD_COND_INITIALIZER;
SPool TS(1);                 // one worker thread

void io_thread(void *idp)    // worker thread code
```

Continued

```
    {
    Timer T;
    T.Wait(2000);     // perform IO
    std::cout << "\n IO task done. Signaling" << std::endl;

    pthread_mutex_lock(&flock);    // toggle predicate
    flag = false;
    pthread_mutex_unlock(&flock);
    pthread_cond_signal(&fcond);   // signal change
    return(NULL);
    }

 int main(int argc, char **argv)
    {
    flag = true;
    TS.Dispatch(io_thread, NULL);

    // – – – – wait for false – – – – – –
    pthread_mutex_lock(&flock);
    while(flag==true)
       pthread_cond_wait(&fcond, &flock);
    pthread_mutex_unlock(&flock);
    // – – – – – – – – – – – – – – – – – –

    std::cout << "\n From main: IO operation completed "
              << std::endl;
    TS.WaitForIdle();
    return 0;
    }
```

LISTING 6.6

IoTask_P.C

In this code flag is the boolean variable used as predicate, flock is the mutex protecting flag, and fcond is the condition variable used to signal that flag has been toggled. The main() function initializes flag to true, launches a worker thread, and waits for false. The worker thread is first blocked for 2 s (this is the I/O operation). Then, it toggles the predicate *with a locked mutex* and signals the event.

Examples 1: IoTask_P.C, IoTask_S.C, IoTask_W.C

To compile, run make iotask_p, iotask_s or iotask_w. This example exhibits event synchronization between the master and a worker thread.

6.6.2 TIMED WAIT IN C++11

Since the C++11 interfaces discussed above for timed waits are not self-evident, an example follows clarifying their meaning. The idea is simple. A worker thread is launched to perform a lengthy I/O operation, emulated by sleeping for 5 s. The main() function, rather than performing an indefinite wait for the event, only waits for a succession of 1-second intervals until the expected event occurs. Running the example, it is observed that the wait is timed out four times, as expected, before the event occurs.

The wait_for() version that incorporates the predicate will be used to implement the timed wait, because this makes life much simpler. Indeed, when dealing with a timed wait there are *two* things to be checked:

- Has the function returned because the wait has been timed out, or because the event has been signaled?
- In the last case, is it a real or an spurious wakeup? This is the reason behind the while loop that checks the predicate again.

The wait_for() version that incorporates the predicate merges internally these two checks and returns a unique answer: true if the event has been signaled *and the predicate is true*, and false in case of timeout. An auxiliary function must be provided, which returns the value of the predicate. This function is called MyPred() in Listing 6.7.

```
using namespace std;

condition_variable CV;
mutex my_mutex;
bool predicate = false;

bool MyPred()
   {
   // This function is always called with the lock owned by the
   // waiting thread locked, according to C++11 documentation
   // ----------------
   bool retval = predicate;
   return retval;
   }

void worker_thread()
   {
   chrono::duration<int, milli> delay(5000);
   this_thread::sleep_for(delay);
      {
      unique_lock<mutex> lock(my_mutex);
      predicate = true;
      }
   CV.notify_one();
   cout << "\n I/O operation terminated " << endl;
   }

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

Continued

```
bool retval;
chrono::duration<int, milli> delay(1000);
thread T(worker_thread);
T.detach();
do
   {
   unique_lock<std::mutex> my_lock(my_mutex);
   retval = CV.wait_for<int, milli>(my_lock, delay, MyPred);
   if(!retval) cout << "\n Timed out after 1 second" << endl;
   }while(!retval);
cout << "\n Wait terminated " << endl;
}
```

LISTING 6.7

TimedWait_S.C

The worker thread sleeps for 5 s, sets the predicate to true, and notifies the change. The main thread enters a do loop; each loop iteration waits for 1 second and continues if the wait return value indicates a timeout. One interesting point in this code is the MyPred function, which reads the predicate and returns its value. One could imagine that reading the predicate should be done with the locked mutex. This is not, however, needed. The C++11 documentation explicitly indicates that the predicate function to wait_for() *is always called internally with a locked mutex*. See the C++11 documentation annex in [14].

Example 2: TimedWait_S.C

To compile, run make twait_s. This example exhibits event synchronization between the master and a worker thread.

6.6.3 BARRIER SYNCHRONIZATION

Barrier synchronization is probably one of the most commonly used synchronization tools in application programming, particularly in codes implementing a data parallel context in which different threads are performing the same operations on different subsets of a global data set. The computation of a scalar product of two vectors performed in Chapter 5 is a simple example of a data parallel program.

There are many iterative data parallel algorithms in which the same computational workload must be repeatedly applied to the global data set until convergence is achieved. When this happens, it is necessary to make sure that all threads sharing the computational workload are always working at the same iteration level. The original algorithm will not be respected if a thread starts manipulating global data at the (N+1)st iteration while some other threads have not yet completed the Nth one. This requires a *barrier synchronization* at the end of each iteration. At this point, all cooperating threads must wait for those that are not yet finished, before continuing with the next iteration. Chapters 13–15 will show several barrier synchronizations of this kind in action.

Barrier synchronization can be implemented in a simple way. The main idea is to introduce a shared counter whose initial value is the number of cooperating threads. When a thread reaches the synchronization point, it decreases the counter and waits on a condition variable if the counter has not reached zero. The synchronizing event happens when the last thread finally decreases the shared counter to zero. At this point, this thread does not wait: it notifies the other waiting threads that the event they are waiting for has taken place, and that they can all resume execution.

The Pthreads and Windows libraries do have a barrier primitive, and OpenMP has a barrier directive. The simple example that follows demonstrates how a barrier synchronization code operates. A set of Nth threads is launched: All threads write a "before" message, wait on the barrier, write an "after" message, and exit. The code works properly if, no matter how many threads, all "before" messages precede the "after" messages.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <SPool.h>

int     count, Nth;
pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond  = PTHREAD_COND_INITIALIZER;
SPool *TS;

// Auxiliary function called by worker threads
// -------------
void BarrierWait()
   {
   // -----------------
   // Acquire mutex and decrease count. If count>0, wait on
   // condition. If count==0, print ID and broadcast wake up.
   // -----------------
   pthread_mutex_lock(&count_lock);                    // LOCK
   count--;
   if(count)
      {
      while(count)
          pthread_cond_wait(&count_cond, &count_lock);
      pthread_mutex_unlock(&count_lock);               // UNLOCK
      }
   else
      {
      printf("\n Broadcast sent by last thread \n\n");
      pthread_cond_broadcast(&count_cond);
      pthread_mutex_unlock(&count_lock);               // UNLOCK
      }
   }

// Worker threads code
// -------
void worker_thread(void *idp)
```

Continued

```
    {
    int my_rank = TS->GetRank();
    printf("Thread %d before barrier\n", my_rank);
    BarrierWait();
    printf("Thread %d after barrier\n", my_rank);
    }
int main(int argc, char **argv)
    {
    if(argc==2) Nth = atoi(argv[1]);
    else Nth = 2;
    count = Nth;

    TS = new SPool(Nth);
    TS->Dispatch(worker_thread, NULL);
    TS->WaitForIdle();
    return 0;
    }
```

LISTING 6.8

Simple barrier code, Pthreads version

In the code above, the main thread does nothing beyond initializing data, and launching the parallel job. It does not participate in the barrier synchronization, an affair involving only the worker threads. The number of worker threads is obtained from the command line (the default is 2), and main initializes the thread pool SPool object with the required number of threads. The main thread also sets initially the counter integer, which is the predicate in the barrier wait, to the number of threads.

In the thread function listed above, each worker thread prints a message before and after calling the BarrierWait() function, which encapsulates the whole barrier synchronization pattern between the worker threads. Listing 6.8 shows how the BarrierWait() function implements the algorithm previously described:

- When the function is called, it starts by locking the mutex and decreasing the counter.
- If the counter is not zero, the thread goes to wait *with the locked mutex*, according to the wait protocol.
- If the counter is zero, the thread does not wait. It prints a broadcast message, then it broadcasts the condition that will wake up all the waiting threads, unlocks the mutex, and returns.

Examples 3: Barrier_1_P.C, Barrier_1_S.C, Barrier_W.C

To compile, run make barr1_p, barr1_s or barr1_w. The number of worker threads is read from the command line (the default is 2).

When running the code one observes that, no matter how many worker threads are launched, there are first the “before” messages of all threads, then the broadcast message of the last thread, and finally the “after” messages of all threads.

Problem: this barrier cannot be reused

This code is admittedly very simple. In fact, it is too simple to be useful in general. The reason is that we have constructed a one-shot, disposable barrier. There is no way to call the same BarrierWait() function later on in the same code, simply because the worker threads have no way of resetting the counter flag to its initial value Nth when the threads are released.

Indeed, one could imagine that the thread that arrives last and decreases the counter to zero, could instead reinitialize it to the initial value. But this does not work, because the wait protocol forces threads to check again the predicate counter==0 when they wake up, before breaking away from the while() loop. If they find at wake up a non-zero counter, they go back to sleep. Threads never emerge from the barrier, and the code deadlocks.

Solution: a reusable barrier

The BarrierWait() function can be modified, so as to make it reusable. I first learnt the elegant solution to this problem in D. Butenhof's book, *Programming with POSIX Threads* [11]. It exhibits the flexibility induced by the fact that the condition variable knows nothing about the event to which it is associated, and that programmers are free to choose the predicate describing an event in the most convenient way.

The basic idea is therefore to replace the counter predicate by a more convenient way of describing the event. Any boolean expression that toggles between false and true qualifies as a predicate. This program is therefore modified by adding a new boolean variable called bflag, coupled to the counter in the following way:

- When counter reaches zero, the last thread resets counter to Nth *and toggles the boolean* bflag.
- On successive barriers, bflag will be toggled from true to false to true and so on. *The predicate for the condition variable wait is not the actual value of* bflag, *but the fact that* bflag *has been toggled.*

Now, a thread that has called BarrierWait() can test for the change in bflag in a simple way: it copies the initial value of bflag to a local variable my_flag, and enters a wait with the predicate my_flag==bflag. When it wakes up, it checks if this is still true. Listing 6.9 shows the new, modified, reusable BarrierWait() function. In the client code, the threads execute two consecutive barriers, and everything works correctly. No matter how many threads cooperate in the barrier, the messages they send to stdout are well separated.

```
#include <mutex>
#include <condition_variable>
#include <SPool.h>

int      count, Nth;
bool     bflag;
SPool    *TS;
std::mutex count_mutex;
std::condition_variable count_cond;

// Auxiliary function called by worker threads
// ------------------------
void BarrierWait(int R)
```

Continued

```
    {
    std::unique_lock<std::mutex> lock(count_mutex);
    bool my_flag = bflag;
    count--;
    if(count)
       {
       while(my_flag == bflag)
           count_cond.wait(lock);
       }
    else
       {
       predicate = !predicate;
       count = Nth;
       printf("\n Broadcast sent by thread %d\n\n", R);
       count_cond.notify_all();
       }
    }

 // -----------
 // Worker threads code
 // -----------
 void worker_thread(void *idp)
    {
    int my_rank = TS->GetRank();
    printf("Thread %d before first barrier\n", my_rank);
    BarrierWait(my_rank);
    printf("Thread %d before second barrier\n", my_rank);
    BarrierWait(my_rank);
    printf("Thread %d after all barriers\n", my_rank);
    }

 // ----------- --------
 // Main, same as in previous listing...
 // --------------------
```

LISTING 6.9

Reusable barrier code, C++11 version

Example 4: Barrier2_P.C, Barrier2_S.C, Barrier2_W.C

To compile, run make barr2_p, barr2_s or barr2_w. The number of worker threads is read from the command line (the default is 2).

Comments

In programming with threads, it is good practice to plan for the worst even if the worst is very unlikely. Therefore, let us imagine that there is a large number of worker threads and two very close barriers, as in Listing 6.9. Consider what may happen soon after the threads in the first barrier are released. They are released one after the other, and it is possible that the first released threads reach the second barrier and call again BarrierWait() *before* other threads have emerged from the first wait and checked again their predicate in the first barrier call. Therefore, *the same predicate that allows the threads to emerge from the wait in the first barrier must be able to put them to wait in the next barrier call.* You should convince yourself that this is indeed the case, and that the above barrier wait algorithm is robust.

In Chapter 9, C++ classes will be introduced that encapsulate the barrier code discussed above, as well as other alternative barrier algorithms. Object instances of these classes are easy to use high-level barriers. Remember that Pthreads and Windows have already easy to use barrier utilities, but this is not the case in C++11 or TBB. OpenMP disposes of a barrier directive that will be discussed in Chapter 10.

6.7 C++11 FUTURES AND PROMISES

The C++11 thread standard introduces some additional event synchronization utilities that are very useful in application programming. Somewhat more restricted than condition variables, they are easier to use and very efficient in some specific contexts. These utilities are defined in the <future > header.

6.7.1 STD::FUTURE<T> CLASS

Instances of this class are used to represent *one-off* events, i.e., unique events that will happen only once in the future. This is the restriction with respect to condition variables. Indeed, the reusable barrier example has shown that a unique condition variable can serve to synchronize an unlimited number of future events. Futures, instead, cannot be reused. On the other side, the interesting point is that future objects, besides implementing the synchronization with the one-off event, can be directly used to recover data values returned by it. Future objects transfer data values across threads, and the template parameter T is the related data type. This template parameter must be set to void if the object is used only for synchronization.

An immediate application of this utility is the recovery of a return value from an asynchronous task executed by another thread. When discussing thread function signatures, we observed in Chapter 3 that, contrary to Pthreads and Windows, C++11 thread functions do not return a value. The future class solves this problem, as shown in the example below.

```
include <iostream>
#include <chrono>

int MyFct(int n)
    { return (42+n); }

int main()
    {
    // asynchronous function call:
    // – – – – – – – – – – – – – – –
    std::future<int> retval =
         std::async(std::launch::asynch, MyFct, 2);
```

Continued

```
    std::chrono::milliseconds ms(3000);
    std::this_thread::sleep_for(ms);    // wait 3 seconds
    std::cout << "\nThe value returned is "<<retval.get() << std::endl;
    }
```

LISTING 6.10

ExFuture_S.C

A simple function MyFct is defined, taking an integer argument and returning another integer value. In this example, the future event is the asynchronous execution and termination of this function. The future object retval is initialized by the return value of the std::asynch() function, discussed below. This function, which received as arguments a pointer to the function to be executed, as well as the argument to be passed to it, returns immediately *before* the function is executed. Then, main() waits for 3 s before recovering the return value by a call to the get() member function of the future object.

Example 5: ExFuture_S.C

To compile, run make exfuture.

std::asynch() function

This template function is one of the three possible ways of associating a task with a future.

- It operates exactly like the std::thread constructor. Besides the first argument discussed next, it receives the function pointer and as many other arguments as needed to call the function. The function pointer and the arguments that follow are copied to some internal storage before the function is called, and arguments are then passed by value. If arguments must be passed by address, the std::ref() qualifier must be used, as was the case of the thread constructor in Chapter 3.
- The first argument is a *policy* argument. The std::asynch function may either launch a new thread to run the function, or run it in the current thread and defer the function call to a later time, when the future return value is requested. The argument passed in the example above forces the asynchronous execution in a new thread. Passing std:: launch::deferred, the function call is deferred in the current thread. This first argument can also be ignored, in which case the function behavior is implementation dependent: one of the two policies is applied.

Basic std::future member functions

Very much like the std::thread objects discussed in Chapter 3, std::future objects are not copyable. Like thread objects, a future object can be moved to another one, transferring ownership of the expected event. The moved object becomes invalid, no longer associated with an asynchronous result.

Here are the most relevant member functions of the future<T> class:

BASIC FUTURE<T> INTERFACES

std::future<T> F;

– Default constructor.
– Defines a future not yet associated to an explicit task.

bool valid()

– Checks if the future is associated with an asynchronous result.
– Returns true if this is the case, false otherwise.

void wait()

- If the associated state contains a deferred function, invokes the deferred function and stores the return value.
- Otherwise, it blocks until the asynchronous result associated with *this is ready.

template <typename Rep, typename Period>
future_status wait_for(std::chrono::duration<Rep, Period> const& delay)

- Waits until asynchronous result is ready, or until the time delay is elapsed.
- Returns std::future_status_ready in the first case.
- Returns std::future_status_timeout in the second case.
- Returns immediately if a deferred function call has not yet started execution.
- Return value in this last case is std::future_status_deferred.

T get()

- If the associated state contains a deferred function, invokes the function and returns the return value.
- Otherwise, it blocks until the asynchronous result associated with *this is ready, and then returns the value.

To provide another simple example, the way a thread can wait for a I/O operation is examined next, using futures instead of condition variables. The worker thread executing the asynchronous operation will just wait for 4 s, to simulate a long I/O operation, writing appropriate messages before and after the wait. The main() function uses a std::future<void> object to wait for the event.

```
void io_thread()     // worker thread
   {
   std::chrono::milliseconds ms(4000);
   std::cout << "\n IO thread waiting for four seconds" << std::endl;
   std::this_thread::sleep_for(ms);
   std::cout << "\n IO operation done" << std::endl;
   }

int main(int argc, char **argv)
   {
   std::future<void> FT = std::async(io_thread);  // dispatch async fct
   FT.wait();                                      // wait for event
   std::cout << "\n Main terminates" << std::endl;
   }
```

LISTING 6.11

Io1_S.C

In the next example, the asynchronous operation is the same, but the main() function, rather than performing an indefinite wait for the event, only waits for a succession of one second intervals until the expected event occurs. Running the example, one observes that the wait is timed out four times, as expected, before the event occurs. Notice how much easier it is to program this example with futures than with condition variables.

```
void io_thread()
   { // as in previous listing }

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

Continued

```
    std::chrono::milliseconds ms(1000);
    std::future<void> FT = std::async(io_thread);  // dispatch

    std::future_status status;
    do
       {
       std::cout << "\n Main waits for 1 second" << std::endl;
       status = FT.wait_for(ms);
       } while (status == std::future_status::timeout);
    std::cout << "\n Main terminates" << std::endl;
    }
```

LISTING 6.12

Io2_S.C

There are a few other std::future features worth keeping in mind. There are naturally wait_until() member functions using absolute dates rather than time durations. It is also possible to associate several threads to a “one-off” event. This cannot be done with the original std::future class, which models unique ownership of the asynchronous result. C++11 introduces the std::shared_future class, which is now copyable, so that ownership can be shared. Several threads can have each a proprietary copy of a shared future object, all associated to the same event. A return value can in this way be broadcasted to several threads waiting for the result.

Examples 6 and 7: Io1_S.C, Io2_S.C

To compile, run make io1_s and io2_s.

6.7.2 STD::PROMISE<T> CLASS

Dispatching an asynchronous task with the std::async() function is not the only way of associating a future object with an asynchronous result. There are two others: promises and packaged_tasks. We discuss next the way a std::promise<T> object sets a data item—of type T—which can later on be read with an associated std::future<T> object. The pair promise-future works as follows:

- First of all, a global std::promise<T> object is declared.
- The client thread that wants to recover later on the data item will do two things:
  - Construct the appropriate future object, initialized by a call to the get_future() member function of the promise object. Indeed, it is the promise object that provides the related future object.
  - Call, when the time comes, the get() function of the future object to receive the expected result.
- The worker thread that computes the expected data value sets, when the time comes, the value of the promise by calling the set_value() member function. In so doing, the future becomes *ready* and can retrieve the stored value.

This mechanism is shown in Listing 6.13, where the client thread performs an unlimited wait for an emulated I/O operation. Since the transfer of a data value is needed to implement the client-worker synchronization, a constant integer value is exchanged among the two threads. In this approach, the worker thread that executes the asynchronous task is launched in the usual way. The worker thread is detached because there is no need to join it to observe the end of the I/O task.

```
std::promise<int> P;

void io_thread()
   {
   std::chrono::milliseconds ms(4000);
   std::cout << "\n IO thread waiting for four seconds" << std::endl;
   std::this_thread::sleep_for(ms);
   std::cout << "\n IO operation done" << std::endl;
   P.set_value(13);        // this makes the associated future ready
   }

int main(int argc, char **argv)
    {
    std::future<int> FT = P.get_future();   // get future from promise

    std::thread T(io_thread);   // launch worker thread
    T.detach();

    int retval = FT.get();      // wait for future event
    std::cout << "\n Main terminates with value " << retval << std::endl;
    }
```

LISTING 6.13

Pio1_S.C

Finally, the previous example is reconsidered: now, main() performs successive 1-second timed waits for the end of a 4-second I/O operation, to show the flexibility of this approach. Getting the data value—which is not relevant in this case—is not really needed. The client thread can perform timed waits as before on the future object, and the waited event is the fact that the future is marked as ready when the data value is stored in the promise.

```
std::promise<int> P;

void io_thread()
   { // as in previous listing }

int main(int argc, char **argv)
    {
    std::future<int> FT = P.get_future();   // get future from promise

    std::thread T(io_thread);       // launch worker thread
    T.detach();
```

Continued

```
    std::chrono::milliseconds ms(1000);
```

```
std::future_status status;
do
   {
   std::cout << "\n Main waits for 1 second" << std::endl;
   status = FT.wait_for(ms);
   } while (status == std::future_status::timeout);

std::cout << "\n Main terminates " << std::endl;
}
```

LISTING 6.14

Pio2_S.C

Examples 8 and 9: Pio1_S.C, Pio2_S.C

To compile, run make pio1_s and pio2_s.

Hopefully, this discussion is sufficient to convince C++ programmers of the substantial interest of the extended C++11 synchronization utilities. There are many interesting subjects not discussed here. If, for example, a promise is not fulfilled because the promise object is destroyed without setting the data value, an exception is stored instead. The client thread may in this way be informed that something is not correct. C++11 has a very efficient way of transferring exceptions among threads. All these subjects are discussed extensively in [14].