CHAPTER

CREATING AND RUNNING THREADS

3

3.1 INTRODUCTION

In high-level programming environments—like OpenMP or TBB—threads are implicit, and the basic thread management operations discussed in the first part of this chapter are silently performed by the runtime library. Most of the examples in this book, indeed, use OpenMP, TBB, or the high-level utilities of the vath library. Nevertheless, understanding the way basic libraries set up a multithreaded environment is useful knowledge, which definitely contributes to the main purpose of this book: learning to think about threads. It may be useful, in a specific parallel context, to launch one or more threads to manage a special issue, like a long network connection, and it is good to know how this can be done.

The rest of the chapter is organized as follows:

- First, an overview is proposed of the Pthreads, Windows, and C++11 interfaces for the creation of a team of worker threads, in order to perform some parallel treatment. A few simple examples are given in each case.
- Next, a C++ class from the vath library is introduced that provides a direct and user friendly way of encapsulating the native libraries' programming interfaces. A number of simple examples are presented.
- Then, anticipating the discussion of Chapter 10, a first look is taken at OpenMP, focusing on those basic features needed to implement an OpenMP version of the previous examples. Our purpose is to underline the universality of concepts and similarities between different approaches.
- Finally, to justify the interest in understanding different programming environments, an example of an application emulating a database search is proposed. This example is easily implemented using features from the basic libraries that were not available in OpenMP before the recent 4.0 release.

3.2 OVERVIEW OF BASIC LIBRARIES

Based on their functionality, the utility functions proposed by the basic libraries can be classified as follows:

- Thread management functions, dealing with the issues related to thread creation, termination, and return. They are discussed in this chapter.

- Thread synchronization functions, a critical set of library functions dealing with the basic synchronization mechanisms for threads. They are discussed in Chapters 5 and 6.
- Miscellaneous services, specialized services, often useful in application programming:
  - Function calls dealing with thread identities, and with initialization issues in library routines, useful when the number of threads engaged in its execution is not known at compile time. This subject is introduced in this chapter.
  - Thread-local storage, providing a mechanism for allowing threads to keep a private copy of dynamically allocated local data. This feature is extensively discussed in Chapter 4.
  - Thread cancel functions. Used for enabling a thread to terminate (kill) another thread. This delicate feature is rarely used in application programming. It is sketched in the final example of this chapter.
- Thread scheduling functions allowing the programmer to determine the priority level of a new thread. By default, threads are democratically scheduled with equal priorities, and this option is very rarely modified in application programming. This feature, used mainly in system programming, is far too advanced to be discussed in this book.
- Signal handling functions. They deal with the interaction of threads and signals in inter-process communications. Again, this subject is beyond the scope of this book.

Various aspects of Pthreads, Windows, and C++11 programming are therefore discussed in this and the next chapters, concerning the basic issues of thread management and synchronization, and providing a number of examples as well as portable application codes.

3.3 OVERVIEW OF BASIC THREAD MANAGEMENT

Most basic concepts and features are common to all basic libraries. They all make extensive use of a number of specific data items to create, manage, and synchronize threads. Several new *data types* are therefore introduced. They may have a slightly different look and feel in Pthreads, Windows (C libraries), and in C++11, but their behavior is very similar. In Pthreads and Windows, they are opaque structures whose inner content is not directly relevant to the programmer. In C++11, they are, naturally, C++ objects. These data types represent the threads themselves, as well as other objects whose purpose is to implement the different basic mechanisms for thread synchronization.

In order to create a thread a *thread function* must be specified, which is the function to be executed by the new thread during its lifetime. Different native libraries adopt different signatures for the thread function. The library function called to create the thread receives as one of its arguments, a pointer to the thread function to be executed by the new thread.

Moreover *thread attributes* must be specified, defining the way in which the new thread is asked to behave under certain circumstances. The most relevant thread attributes, common to all native libraries because they correspond to generic properties of threads, are:

- The size of the stack attached to the thread.
- The way the thread behaves when the thread function it executes terminates. *Detached threads* just disappear after restoring their resources—like the stack—to the system. *Joinable threads* wait to

be contacted (joined) by another thread to which they may eventually transfer a thread function return value.

- The way a thread reacts when killed by another thread: either it stops immediately or it defers cancellation until it reaches a predefined place in the instruction flow.
- The scheduling properties, which, as explained above, allow the programmer to determine the priority level of the new thread.

This information must be passed to the function call that creates the thread. Pthreads packs all the different thread attributes in a unique thread attribute data item, which is then passed to the thread creation function. Windows does not pack thread attributes; they are all passed as distinct arguments to the thread creation function. In C++11, thread attributes are private data items in the C++ object that represents the thread. All thread creation functions accept reasonable default values for the thread attributes, if they are not specified in the thread creation function call. We will focus in the rest of this book on the first two attributes, which are the most relevant ones for applications programming, and ignore the others.

Stack size.

The default stack size may need to be increased if the thread function and/or the functions it calls create too many local variables, thereby producing a stack overflow. But remember that the default stack size is platform dependent, and therefore not portable. It may happen that a stack overflow that occurs in some systems does not occur in others.

Joining threads.

As explained above, when a thread finishes executing its instruction flow, there are two possibilities. Either the thread is detached, in which case it terminates and restores its proprietary resource (the stack) to the system, or it is joinable and does not disappear before being contacted by another thread. Obviously, this is in all cases a synchronization event: because the joining thread knows, when the time comes, that the joined thread has terminated, it can rely on this information to proceed accordingly (for example, using results provided by the joined thread). In Pthreads, the detached or joinable status of a thread is a permanent attribute established when the thread is created. We will see later on that C++11 and Windows do not rely on thread attributes to handle this feature.

Comments on error handling

Library functions report errors if something goes wrong. But internal errors coming from an incorrect operation of the library itself are extremely unlikely, and if they happen the only reasonable thing to be done is to abort the program. Most of the time, library errors are generated because library functions access corrupted data, like for example accessing objects that have previously been destroyed.

In multithreading programming the most dreadful errors are those arising from an incorrect control of thread synchronization. These kinds of errors are called *race conditions* because the program outcome depends on the non-reproducible order on which asynchronous threads access a given data item. These programming errors are not reported by the library functions, and the best possible protection is to make sure they do not happen, following the protocols and best practices discussed in the forthcoming chapters. Some programming environments propose auxiliary tools that are capable of detecting race conditions. Avoiding race conditions is the subject of Chapter 5.

3.4 USING POSIX THREADS

The Pthreads library has been available since the early days of the Unix operating system, and there is a large number of bibliographical references, mainly on the Internet. Two classic books on this subject are “Programming with Posix Threads” by D. Butenhof [11], and “Pthreads Programming” by B. Nichols, D. Buttlar, and J. P. Farrel [12]. A good support is also provided by the Unix-Linux `man` pages, and a good starting point to browse the manual documentation is to run `man pthreads`.

3.4.1 PTHREADS ERROR REPORTING

Traditional Unix and C language conventions on error status reporting rely on library functions returning an integer value. A function call that succeeds returns a non negative integer. Programmers can then use positive return values to return some useful additional information, or return zero if all is needed is to report a successful function return. On failure, ordinary library functions return the special value (−1) and set the global value `errno` to whatever error code needs to be reported. Users must therefore monitor the function return value and take special action—like checking the value of **errno**—if the return value is (−1).

Checking the value of a global variable in a multithreaded environment is not at all appropriate. If a thread detects a (−1) as a return value and decides to check `errno`, there is no way to know if the `errno` value is the relevant one, or if it has been subsequently modified by another thread since the initial error condition occurred. Therefore, the Pthreads library adopts another convention for error reporting that does not rely on testing a global variable:

- Pthreads functions *always return the value 0 on success.*
- If the function call is not successful, then the *positive* error codes defined in `errno.h` are transferred to the caller via the function return value.
- Pthreads library functions never return directly any useful information other than error reporting. Another kind of information is returned via pointer types passed as function arguments.
- If one wants nevertheless to transfer useful information to the caller through return values, *negative integers* can be used. A negative return value *cannot* be an error code. We will use this option in our later discussion of synchronization objects.

Conventions adopted.

In principle, for each library function call one should check the return value. However, in case of error there is little that can be done other than aborting the program. In order to avoid exceedingly verbose listings, error checking is suppressed in the listings copied in the text of this book, ignoring the return values. The code below shows how a function call looks in the source code:

```
//Pthread function call in text:
//- - - - - - - - - - - - - - - - -
pthread_XXXX(arguments);

//Pthreads function call in sources
//- - - - - - - - - - - - - - - - - -
int status = pthread_XXXX(arguments)
if(status)
   {
   fprintf(stdout, "Error in pthread_XXXX");
   exit(0);
   }
```

LISTING 3.1

Conventions on error checking

3.4.2 PTHREADS DATA TYPES

Pthreads introduces a number of new data types. They are named following the generic form identifier_t: some characteristic name that unambiguously describes the type and the role of the variable, followed by the underscore appending a t at the end. This convention is traditionally adopted in C to introduce user-defined data types.

> Pthreads data types are OPAQUE data objects. They can be seen as C structures whose internal implementation is hidden to the programmer. These opaque structured are accessed via library functions. Their addresses are passed to the library functions that operate on them. Programmers writing portable code should make no assumption about their precise representation.

For example, pthread_t is a thread identifier, a kind of identity card that uniquely identifies a thread during its lifetime. This identifier is initialized by the function pthread_create() that creates the thread. Other data types are, for example, pthread_mutex_t, a data structure used to control concurrent access to shared data, and pthread_cond_t, a data structure used to implement event synchronization of threads. They are discussed in Chapters 5 and 6.

Some Pthreads data types identify thread specific objects—like threads, mutexes, or condition variables. Other Pthread data types define *attributes* of these objects—like pthreads_attr_t—which encapsulate a set of specific thread attributes defining their behavior under certain circumstances. They are set by the programmer before the thread is created, and the attribute variable address is passed to pthreads_create(). Attribute data types are introduced with the purposing of adding new functionality to the library without changing the existing API.

Even if a data item of type pthread_t is, in most systems, an unsigned integer, well-crafted code should not rely on this fact. The precise content of a given data type depends on the specific Pthreads implementation, and this is, naturally, a highly non-portable issue. However, the Pthreads library includes a number of accessory functions providing a portable interface to the inner content of the opaque objects. Data types values should therefore be initialized and modified by following the Pthreads specifications. This means using library provided interface functions for reading or modifying attribute values, and, in some cases, using predefined values provided by the library for their initialization. The examples developed later on will clarify the manipulation of attribute data types.

3.4.3 THREAD FUNCTION

The thread launching a new thread calls the pthread_create() library function and passes to it, among other things, *a pointer to the function that the new thread must execute*. This thread function is required in Pthreads to have a well defined signature:

(void *) th_fct(void *P); (function name is arbitrary)

It is therefore necessary to provide a function that receives a void* and returns a void*. The Pthreads library adopts this signature to allow data to be passed from the parent thread to the children threads at start-up, and from the worker thread back to a joining thread at termination, as will be discussed next. Indeed, a void* is a universal container in C: the address of *any* data structure can be cast to a *void** and passed to the thread function. This function will, when the time comes, recast it to a pointer to the original data type, before using it. In the same way, the thread function can allocate any data structure and return its address as a void* to the joining thread.

One should be aware that passing data back and forth to a new thread in this way is not really mandatory. Indeed, global variables are shared by all threads in the process, and threads can communicate by writing and reading them. However, since threads are asynchronous, this requires explicit synchronization in order to make sure the writes and reads occur in the expected order. Therefore, this simple, built-in way of passing information to a new thread without explicit synchronization is useful, and this feature will sometimes be used in the rest of the book. C++11 does not support this feature: its thread functions return void. C++11, however, introduces an additional synchronization mechanism enabling the programmer to retrieve a return value from a terminating thread, which will be described in Chapter 6. Chapter 9 proposes a portable utility that enables the safe transfer of data values among threads.

3.4.4 BASIC THREAD MANAGEMENT INTERFACE

Here is the signature of the functions used for basic thread management in Pthreads:

```
int pthread_create(
       pthread_t              *thread,
       pthread_attr_t         *attr,
       (void *) (*thread_fct) (void *),
       void                   *arg);

int pthread_join(
       pthread_t           *thread,
       void                **value_ptr);
```

LISTING 3.2

The basic Pthreads thread management functions

Consider first the pthread_create() function, which seems to have a rather complicated signature. Four arguments are passed to this function:

- A pointer to a thread identifier. This is an *output* parameter: the function receives the address of an uninitialized data item and, when it returns, the thread identifier has been initialized and henceforth identifies the thread just created.
- A pointer to a thread attribute variable. This is an *input* parameter. This thread attribute has been previously initialized to whatever attributes are required for the new thread. If the NULL value is passed, the library adopts the set of default attributes for the thread.
- A pointer to a function thread_fct(), i.e., the thread function. This is the function the thread will execute during it lifetime. It must, of course, be declared and defined somewhere else in the code, and its address is passed here. Remember that, in C-C++, the address of a function is just its name.
- A void pointer arg that is precisely the argument to be passed to the thread function thread_fct().

Notice that, when this function is called, the thread function is not called directly. This function simply passes to the Pthreads library the information needed to create the new thread. The thread function will be called by the Pthreads library after the execution environment of the new thread (in particular, the new stack) has been created and installed in the application.

3.4.5 DETACHMENT STATE: JOINING THREADS

It was stated before that a thread created as *detached* just disappears when the thread function terminates and the eventual return value of the thread function is ignored. But a *joinable* thread needs to make contact with another thread to really terminate its life cycle.

When a thread is created, the Pthreads library returns a unique thread identifier of t type pthread_t. Any other thread that knows the identifier ID of the target thread whose termination needs to be verified can make the following call:

```
void *P;
...
pthread_join(ID, &P);    // wait until thread ID terminates
```

LISTING 3.3

Joining a terminating thread

When this call is made by a thread joining another target thread, the following things happen:

- If the target thread has terminated, the join operation takes place immediately: the joining thread eventually recovers the return value, and the target thread finishes its life cycle.
- Otherwise, the runtime code moves the joining thread to a blocked state, *not using CPU resources*. When the target thread terminates, the runtime code wakes up the joining thread, which is then rescheduled, and the join operation takes place.

The above described mechanism is a special case of a generic *event synchronization* mechanism, which will be discussed in detail in Chapter 6 (the event in this case being the termination of the target thread). The target thread void* return value is recovered at the address pointed by P, passed as second argument. Passing NULL as second argument means the joining thread is not interested in the return value, which is ignored. *We emphasize again that this is really a synchronization point among two threads.*

Some miscellaneous issues

There are a few useful Pthreads library functions that propose some miscellaneous services:

- void pthread_exit(). Normally, a thread terminates when its thread function terminates. But a call to this function in some place of the thread function—or in functions called by it—immediately terminates the running thread.
- pthread_t pthread_self(). This function returns the thread identity of the caller thread.
- int pthread_equal(pthread_t t1, pthread_t t2). This function compares two thread identities, and returns 0 if they are different or 1 if they are identical. As stated before, thread identities are opaque objects that cannot be compared directly. An example will be provided showing the utility of these last two library functions.

3.4.6 SIMPLE EXAMPLES OF THREAD MANAGEMENT

A few simple examples of the Pthreads protocol to create a team of worker threads follow. The sources are well commented, and will not be completely reproduced here.

Example 1: Hello1_P.C

Listing 3.4—source in file Hello1_P.C—shows how to create a set of threads that print a message to the screen. Threads are then joined one at a time by the main thread.

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

#define NTHREADS  4

/*-----------------------------
 * Declare an array of thread identifiers. This array
 * will be initialized by pthread_create()
 *---------------------------- */
pthread_t hello_id[NTHREADS];

/*------------------------------
 * This is the thread function executed by each thread.
 * Thread functions receive and return a (void *), but
 * here the argument and return value are ignored
 *----------------------------- */
void *hello_world (void *arg)
   {
   printf ("Hello, world \n");
   return NULL;
   }

int main (int argc, char *argv[])
```

```
{
int i, status;

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 * Create threads. The first NULL means that the new thread
 * picks the default attributes. The second NULL is the void
 * pointer passed as argument to the thread function
 *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
for(i=0; i<NTHREADS; i++)
     pthread_create (&hello_id[i], NULL, hello_world, NULL);

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 * Join the threads in the same order they have been created.
 * The NULL argument means that we do not want to receive the
 * thread return value in the join operation.
 *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
for(i=0; i<NTHREADS; i++)
   pthread_join (hello_id[i], NULL);

printf("\n From main : threads have been joined\n");
return 0;
}
```

LISTING 3.4

Creating threads: Hello1_P.c

Example 1: Hello1_P.C

To compile, run `make hello1p`. The number of threads is 4.

Example 2: Hello2_P.c

In this example, a new feature is added to the code described above. The possibility of passing an argument to the thread function is used to communicate to each worker thread an integer value (its rank). The rank values start from 1, and they correspond to the order in which threads are created. A worker thread can then use its rank value to select the work to be performed through conditional statements. In the present case, each thread prints a message reporting its rank.

This code introduces a global array holding the integer rank values, and passes to each thread the address of the corresponding array element, cast as a void pointer. Listing 3.5 indicates the modifications made to Hello1_P.c

```
int th_ranks[NTHREADS+1];   /* array of integers */

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 * This is the new thread function executed by the threads.
```

Continued

```
 * It now receives an argument, and prints its value.
 *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
void *hello_world (void *arg)
   {
   int rank;
   rank = *(int *)arg;   // read integer value
   printf ("Hello IDRIS from thread %d \n", rank);
   return NULL;
   }

int main (int argc, char *argv[])
   {
   int n, status;

   // Initialize the ranks array
   for(n=1; n<=NTHREADS; n++) th_ranks[n] = n;

   /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    * Create threads. adopting default attributes. The last
    * argument is the rank argument  passed to the thread function.
    *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   for(n=1; n<=NTHREADS; n++)
      pthread_create (&hello_id[n], NULL, hello_world,
                      &th_ranks[n]);

   // The rest of the code is unchanged
   ...
   }
```

LISTING 3.5

Creating threads: Hello2_P.C

Example 2: Hello2_P.C

To compile, run make hello2p. The number of threads is 4.

Example 3: Hello3_P.C

This example does exactly the same as the preceding one, in a way that superficially looks correct but which in fact is not. Rather than using a global array holding the integer rank values, we may try to make things simpler by just passing to each thread the address of the loop counter n incremented by the main thread.

```
int main (int argc, char *argv[])
   {
   int n, status;
/* ------------------------------------
 * Create threads. The code passes to each new thread the
 * address of the loop counter n. This looks correct, but it
 * is not.
 *------------------------------------ */
for(n=0; n<NTHREADS; n++)
   pthread_create (&hello_id[n], NULL, hello_world, (void*)&n);

// The rest of the code is unchanged
...
}
```

LISTING 3.6

Creating threads: Hello3_P.C

Example 3: Hello3_P.C

To compile, run make hello3p. The number of threads is 4.

The first thing to be observed is that the loop counter n—whose address is passed to the thread functions executed by the new threads—is a *local variable* sitting in the stack of the main thread. The new threads will be reading its value via the pointer they receive. This is then one of the first examples of a thread publishing its private data and allowing other threads to read it, as discussed in Chapter 2, which is, technically, a legitimate operation.

However, this code is incorrect, because the implicit assumption has been made that the thread just created is immediately scheduled, and that it immediately reads the loop counter value n *before main() has increased it to launch the next thread*. This is just not true: threads are asynchronous, and no reasonable assumptions can be made on the way they are scheduled. When this code is executed, the rank values reported are not predictable. Some rank values may be missing because they were not read before the loop counter is increased, and others may be repeated because they were read more than once.

This is the first example of a *race condition* in which the results of the execution of a code are not deterministic, because they are sensitive to the way threads are scheduled. As we will see, explicit synchronizations are required to avoid race conditions.

Example 4: Intrinsic rank of a thread

In the two preceding examples, a team of worker threads was created by passing explicitly an integer value, corresponding to the order in which threads are created. Notice that this integer rank is also the index of the thread identifier in the global identifier array, called hello_id[] in the examples above.

In the vath utility we will soon discuss, this integer rank is extensively used to identify a thread, and we would like to install a context in which functions called by the primordial thread function can also know the rank of the running thread. The GetRank() function listed below solves this problem.

```
// thread identity array
//– – – – – – – – – – – – –
pthread_t hello_id[NTHREADS+1];

// Auxiliary function that returns the caller
// thread rank
//– – – – – – – – – – – – – – – – – – – – – – –
int GetRank()
   {
   pthread_t my_id;
   int n, my_rank, status;

   my_id = pthread_self();   // who  am I?
   n = 0;
   do
      {
      n++;
      status = pthread_equal(my_id, hello_id[n]);
      } while(status==0 && n < NTHREADS);

   if(status) my_rank=n;    // OK, return rank
   else my_rank = (–1);     // else, return error
   return my_rank;
   }
```

LISTING 3.7

Computing a running thread rank: Hello4_P.C

The basic idea is simple. The function calls pthread_self() to get the identity of the running thread. Then it scans the hello_id[array, comparing the running thread identity with the stored thread identities, until there is a fit. It returns the running thread array index, or (-1) if the caller thread is not a member of the workers team.

The full example is in source file Hello4_P.C. Notice that in this case we no longer need to pass explicitly the integer rank to each worker thread. We have presented this function in the way it is used in the vath library, where thread ranks are in the range $[1, N]$. For this reason, a hello_id[] array of dimension NTHREADS+1 is introduced, and the first slot is not used.

Example 4: Hello4_P.C

To compile, run make hello4p. The number of threads is 4.

Example 5: Retstring_P.C

This example shows how the pthread_join() function can be used by a worker thread to transfer a return value to the joining master thread. The main thread launches one worker thread, which allocates memory, creates a string, and returns its address to the main thread. This example is proposed to illustrate how the return value mechanism works, but this feature will not be used in the rest of the book.

Example 5: Retstring_P.C

To compile, run make retstring. There is a master thread and a worker thread that returns a string to the master.

3.5 USING WINDOWS THREADS

The multithreading support of the Windows operating system has today the same conceptual structure as the support provided by POSIX threads. Library function names may be different, but the basic functionalities are very close. In the early days of Windows NT there was an area—event synchronization of threads—where the protocols and best practices were very different. However, since Windows Vista a number of new features have been introduced in the Windows API that, to a large extent, filled the gap with the POSIX threads programming interfaces.

Today, migrating code to and from Pthreads is much easier. The whole Windows API is accessed through an upper software layer called the CRT (C Runtime Library), which benefits from an excellent online documentation, to which we will systematically refer in this book. In the Windows API home page [13], the most relevant entries for our purposes are in the System services -> Threads and System services->Synchronization links. But we will be more precise in each particular case.

Windows thread data types

As stated before, thread support in Windows is part of the operating system. Thread objects, as well as other synchronization objects, are therefore managed by the Windows kernel. They are identified in Windows by a data type called HANDLE, which is in fact un unsigned long integer. Client code asks the kernel to create a specific object—a thread, a mutex, etc.—and the kernel returns a HANDLE to the object. This handle can be seen as an index in some internal array where the kernel allocates and ranges the corresponding objects. Client code must pass the handle to the kernel when requesting services involving the target object. However, all handles are of the same kind, so it is the programmer's responsibility to keep track of the kind of object that corresponds to a given handle.

3.5.1 CREATING WINDOWS THREADS

The most basic way of creating a thread is a call to the CreateThread() function, which takes several parameters including a reference to the thread function, and returns directly the thread handle. This is not, however, the recommended way of creating a Windows thread, partly because there is no guarantee the thread will already be scheduled when the function returns. This may, in some complex contexts, raise some problems.

Windows has two other high-level C functions that can be called to create a thread: _beginthread() and _beginthreadex(). Their signatures, as well as the signatures of the thread functions they require, are given in Listing 3.8:

```
// Thread function for _beginthread():
//- - - - - - - - - - - - - - - - - - -
void th_fct1(void *data);

unsigned int _beginthread(
       void     (*th_fct1) (void *),                 // thread fct
       unsigned stack_size,                          // stack size
       void     *data);                              // arg of thread fct


// Thread function for _beginthreadex()
//- - - - - - - - - - - - - - - - - - - - -
unsigned int __stdcall th_fct2(void *data)

unsigned int _beginthreadex(
       void     *sec_attr,                           // security attribut
       unsigned (*th_fct2) (void *),                 // thread fct
       unsigned stack_size,                          // stack size
       void     *data,                               // arg of thread fct
       unsigned flag,                                // state flag
       unsigned *thID);                              // thread ID
```

LISTING 3.8

Creating Windows threads

In both cases, the thread function takes, as in Pthreads, a void* as argument. But the first one (for _beginthread()) has no return value, while the second (for _beginthreadex() returns an unsigned int. We will soon see that there is some logic to this, given the way both functions operate. Let us first examine the function arguments. There are many of them, because Windows does not pack thread attributes in a unique attribute data item. But passing in most cases the value 0 (for integers) or NULL (for pointers) selects reasonable default values.

- Return value: Both thread creation functions return the thread handle in the form of an unsigned int. When this handle is passed to library functions that act on the thread, it must be explicitly recast to a HANDLE, as we will see in the examples that follow.
- Arguments:
  - A pointer to the thread function, as well as a void pointer to its data argument, as was the case in Pthreads.
  - A stack size, if the default value needs to be modified. Otherwise, passing 0 picks the default stack size.
  - A void* to a security attribute of the threads. We will always accept the default value. The Windows API documentation can be consulted for further details.
  - A flag (unsigned int) that decides whether the thread starts running immediately or is created in a suspended (blocked) state. In the second case, the new thread starts running when the function ResumeThread() is called, with the corresponding thread handle as argument. Passing the value 0 for this flag creates a thread that starts running immediately.

– A pointer to an unsigned integer, called thID in the listing above, which is an *output* parameter where the kernel returns another integer identifier of the new thread, *different form the thread handle*. Its possible usefulness is discussed below.

Why are there two different thread creation functions?

These two functions are not members of the basic Windows API; they belong to un upper software layer called the CRT (C Runtime Library). The initial purpose of _beginthread() was to have a simple and direct way of creating a thread. This function was extended later because the additional arguments incorporated in _beginthreadex() were needed in many cases. However, besides passing more detailed information, a fundamental difference was introduced in the way these two functions manage the handles of the threads they create. When a thread created with _beginthread() terminates, the kernel will immediately close (release) its handle. This very same handle can eventually be attributed to another thread created later. In this case, the programmer has no control whatsoever on the lifetime of the thread handle. For terminating threads created with _beginthreadex(), the handle must be explicitly closed by the programmer by calling the CloseHandle() function, typically after joining the target thread.

Therefore, in spite of the fact that _beginthread() looks simpler, it has to be used with care. Let us imagine that a thread is created and its handle stored in h1. Later on, a second thread is created and its handle stored in h2. It may happen that, in the meantime, the first thread has terminated, and the kernel has reused the first handle to create the second thread. In this case, h1==h2 refers to the *second* thread, not the first which has already terminated. The usage of _beginthreadex() is recommended, even if threads are not joined, to prevent reusage of handles by the kernel.

Another way of looking at this issue is to say that _beginthread() is safer when dealing with detached threads whose termination is not tracked and, from this point of view, it is natural that its associated thread function has no return value. On the other hand, _beginthreadex() is more naturally related to the creation of joinable threads.

Terminating threads: In the same way that pthreads_exit() terminates a POSIX thread when called by a thread function (or a function directly or indirectly called by the thread function), Windows has two other CRL functions, _endthread() and _endthreadex(unsigned n), that terminate threads created with _beginthread() or _beginthreadex(), respectively.

Sleeping. Windows has a very handy utility: the Sleep(msecs) function, which puts the caller thread in a blocked state for a duration of msecs milliseconds. This utility will be used extensively in our examples.

3.5.2 JOINING WINDOWS THREADS

The Windows API has two functions a thread can call to wait for the termination of a thread, of for the termination of a worker team, whose signatures are given in Listing 3.9.

```
DWORD WaitForSingleObject(
       HANDLE hdl,                // target thread handle
       DWORD  dwMillisecs);       // duration of wait

DWORD WaitForMultipleObject(
```

Continued

```
        DWORD  count                  // size of Hdl array
        HANDLE *Hdl,                  // array of thread handles
        BOOL   waitAll,               // flag that defines action
        DWORD  dwMillisecs);          // duration of wait

// Getting return values
//------------
DWORD  dwRetval;         // return value recovered here
HANDLE h;                // target thread handle
...
GetExitCodeThread(h, &dwRetval);
```

LISTING 3.9

Joining Windows threads

- In this context, DWORD is equivalent to uint.
- WaitForSingleObject():
  - The caller thread waits for the termination of the thread whose handle is passed as an argument.
  - This is a *timed wait*. The caller thread waits up to dwMillisecs milliseconds. It returns the symbolic constants WAIT_TIMEOUT if the wait interval is exhausted, or WAIT_OBJECT_O if the function returns because the target thread terminated before.
  - In order to have an indefinite wait, the symbolic constant INFINITE is passed as a second argument.
- WaitForMultipleObject:
  - Hdl is an array of thread handles, of size count.
  - This function waits for the termination of one or all the threads identified in the handle array Hdl.
  - If waitAll is true, it waits for the termination of all the threads. Otherwise, waits for the termination of the first.
  - The timed wait works as in the previous case.
  - Return values are the same as in the previous case if the function waits for all the threads (the only usage we will make). For the more general case, look at the Windows API documentation.

These wait functions have a broader scope than waiting for thread termination. It was observed before that joining a thread can be seen as a special case of event synchronization: the joining thread waits for an event, the event being the termination of the joined thread. Windows considers thread termination as a synchronization event triggered by the terminating thread. Other Windows synchronization objects also trigger synchronization events, and a thread can wait for them by calling the same functions, but passing the appropriate handles to the other objects. Notice also that, contrary to the join() function in Pthreads, these functions have no reference whatsoever to the thread function return value. In order to recover the return value of a thread created by _baginthreadex(), the third function listed above, GetExitCodeThread(), must be called. It behaves as follows:

- The first argument is the target thread handle, and the second argument is an output parameter where the thread function return value is recovered.
- The return value is the symbolic constant STILL_ACTIVE if the thread is still running when the function is called, in which case the value returned in the second argument is not relevant. Otherwise, the value recovered can be trusted.

It follows that the correct protocol to join a worker thread is first, wait for termination, then get the return value, and finally close the thread handle.

3.5.3 WINDOWS EXAMPLES OF THREAD MANAGEMENT

Example 6: Hello1_W.C

Here is the listing of our first Windows example, Hello1_W.C. A thread is created, using _beginthreadex(), to print a hello message to the screen. Notice that the process.h header must be included when the _beginthreadex() function is used. This example has been tailored to show some of the features and issues discussed above.

```
#include <windows.h>
#include <process.h>
#include <iostream>

using namespace std;

unsigned __stdcall threadFunc(void *p)
   {
   unsigned n = reinterpret_cast<int>(p);
   Sleep(200);
   if(n<0) _endthreadex(0);
   return n;
   }

int main()
   {
   unsigned long th;
   unsigned ID;
   int      retval;
   cout << "\n Enter return value (int) : " << endl;
   cin >> retval;

   th = _beginthreadex(NULL, 0, threadFunc, (void*)retval, 0, &ID);
   if(th==0)
      {
      cout << "\n beginthreadex failed" << endl;
      return -1;
      }
   cout << "\n Thread running " << endl;
```

Continued

```
    for(;;)
       {
       DWORD dwExit;

       GetExitCodeThread(reinterpret_cast<HANDLE>(th), &dwExit);
       if(dwExit==STILL_ACTIVE) cout << "\nThread still running" << endl;
       else
          {
          cout << "\n Thread exit code was " << dwExit << endl;
          break;
          }
       }
    // tell system that we are finished with this thread
    CloseHandle((HANDLE)th);
    return 0;
    }
```

LISTING 3.10

Creating a Windows thread

First, notice the simplicity of the thread function: it just returns the integer received as the argument. However, this thread function terminates normally only if the received argument is positive, in which case it corresponds to an unsigned integer. Otherwise, the thread function terminates with a call to _endthreadex(). The thread function sleeps for 200 ms before terminating, in order to have a reasonable duration for its life span.

The main() function first reads from stdin the user-selected thread return value (so that one can pass either positive or negative values to check the way the program works). Then, the new thread is created. In this example, default values are adopted for the security attribute, the stack size, and the suspended flag. And the integer value read from stdin is passed to the new thread. This value is printed in the hello message.

After creating the new thread, main() waits for its termination in a peculiar way: it enters into a loop that keeps trying to read the return value for as long as the thread is still active. The loop breaks when the thread is no longer active, and the return value is read and printed. Because the target thread has been put to sleep for 200 ms, there are lots of “thread still running” messages. Once the return value is recovered, main() closes the target thread handle.

This way of waiting for the target thread is wasteful for long waits, because the main thread keeps using CPU cycles in the for loop. When, instead, the main thread calls WaitForSingleObject(), it waits in a blocked state not using CPU cycles. This is the correct wait strategy that will henceforth be adopted.

Example 6: Hello1_W.C

To compile, run make hello1_w.

Example 7: Hello2_W.C

This example is the Windows version of the example previously developed in Pthreads: creating a team of worker threads that prints a message indicating their intrinsic rank, and joining the team. The auxiliary function GetRank()—that computes the intrinsic rank of the caller thread—demonstrates the utility of the thread identity returned by _beginthreadex(), different from the thread handle. Indeed, the only way a thread has of knowing its own identity is by calling the GetCurrentThreadId() function, which returns the thread identity as an unsigned integer. Therefore, in order to identify the rank of a running thread this return value must be compared to the thread identities returned when the thread was created. The GetRank() function in Listing 3.11 is identical to the similar functions encountered before in the Pthreads environment.

```
using namespace std;

unsigned       *ID;    // array of thread identities
int            nTh;    // number of threads

int GetRank()          // auxiliary function
   {
   unsigned my_id;
   int n, my_rank, status;

   my_id = GetCurrentThreadId();     // determine who I am
   n = 0; status = 0;
   do
      {
      n++;
      if(my_id == ID[n]) status=1;
      } while(status==0 && n < nTh);
   if(status) my_rank=n;    // OK, return rank
   else my_rank = (-1);     // else, return error
   return my_rank;
   }

unsigned __stdcall threadFunc(void *p)    // thread function
  {
  int rank = GetRank();
  Sleep(500*rank);
  cout << "\n Hello world from thread  " << rank << endl;
  Sleep(500*rank);
  return 0;
  }

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

Continued

```
{
int n;
unsigned long *th;
HANDLE       *hThread;

if(argc==2) nTh = atoi(argv[1]);  // get nTh from command line
else nTh = 4;

th = new unsigned long[nTh+1];    // allocate arrays
hThread = new HANDLE[nTh+1];

ID = new unsigned[nTh+1];
for(n=1; n<=nTh; n++)           // create threads
   {
   th[n] = _beginthreadex(NULL, 0, threadFunc, NULL, 0, &ID[n]);
   hThread[n] = (HANDLE)th[n];
   }

// Wait for threads to exit
DWORD rc = WaitForMultipleObjects(nTh, hThread+1, TRUE, INFINITE);

for(n=1; n<=nTh; n++) CloseHandle(hThread[n]);
cout << "\n Main thread exiting" << endl;
delete [] ID;
delete [] hThread;
delete [] th;
return 0;
}
```

LISTING 3.11

Creating a worker team

Notice that there are three thread related arrays in this example (we keep using offset 1 arrays, always allocating one extra unused slot):

- th[ ]: this is the array of thread handles returned by _beginthreadex() as unsigned long.
- hThread[ ]: this is the previous array recast to the HANDLE type, needed for waiting for threads or for closing the handles. This array is superfluous, because the th array elements can always be recast as handles on the fly, but we found the code is clearer in this way.
- ID[ ], an array that stores the thread identities returned by _begintreadex(). In the code above, this array is global because it is accessed both by main() and the GetRank() functions. The other arrays do not need to be global, because they are accessed only by main().

The main function reads the number of threads from the command line (four threads is the default) and then allocates the three thread related arrays. Then, the worker threads are created in a loop where the three arrays are initialized. Finally, all the worker threads are joined by a call to WaitForMultipleObjects(), with the following arguments:

- nTh, the number of threads, is the size of the hThread array.
- hThread+1 is the starting point of the relevant, size nTh, thread handle array. We are jumping on the first, unused hThread array element.
- TRUE means we are waiting for the termination of all the threads.
- INFINITE means the wait is unlimited.

Once the worker threads have been joined, their handles are canceled. The thread function just gets the thread rank and prints a message. The worker thread lifetimes have been stretched by putting the threads to sleep for durations depending on their ranks.

Example 7: Hello2_W.C

To compile, run hello2_w.

3.6 C++11 THREAD LIBRARY

The C++11 standard has incorporated a complete set of multithreading facilities into the C++ standard library. This evolution is a significant step in software engineering: the possibility of disposing of a portable, low-level programming environment providing practically all the services and features of the native libraries. Developers can use C++11 threads to implement sophisticated system programming with full portability as an extra bonus.

3.6.1 C++11 THREAD HEADERS

When programming in Pthreads, the only header file required to be included is pthread.h. Likewise, Windows only requires windows.h and sometimes process.h, as explained in the previous section. The C++11 thread library, instead, requires a number of specific header files corresponding to the different proposed services, listed below. All the C++11 classes are naturally defined in the std namespace.

- <thread>: facilities for managing and identifying threads, or for putting a thread to sleep for a predetermined time duration.
- <mutex>: facilities implementing mutual exclusion, in order to enforce thread safety in shared data accesses by different threads. This subject is discussed, for all programming environments, in Chapter 5.
- <condition variable>: basic level synchronization mechanism allowing a thread to block until notified that some condition is true or a timeout is elapsed. This subject is discussed, for all programming environments, in Chapter 6.
- <future>: C++11 specific facilities for handling asynchronous results from operations performed by another threads, discussed in Chapter 6.
- <atomic>: Classes for atomic types and operations. Discussed, for all programming environments, in Chapters 5 and 8.
- <chrono>: classes that represent points in time, durations, and clocks. Very complete date-time programming environment, used in this book only to measure execution times.
- <ratic>: compile-time rational arithmetic, not directly used in this book. This facility is needed for the chrono services.

A very complete and detailed discussion of the C++11 threads standard is developed in the book, “C++ concurrency in action” [14]. This book is a basic reference on the subject. Its Appendix D proposes a detailed reference of all the library components.

A few words about implementations. Most widely adopted compilers are today fully—or almost—C++11 compliant. GNU 5.0 or higher compiles all the examples proposed in this book. Visual Studio 2013 has some minor limitations in what coincerns the chapter 4 examples. The GNU compiler C++11 status can be found on the GCC compiler website [15]. Two broadly used programming environments implement the C++11 thread library:

- Boost is fully compliant with the C++11 multithreading standard library. In most cases it is sufficient to replace the std:: namespace qualifier by the boost:: namespace qualifier to match the C++11 standard. Some facilities have different names: the std::chrono header must be replaced by boost::date-time, and the programming interfaces are not exactly the same. In the rest of the book we will adopt the strict C++11 standard and comment about Boost whenever appropriate.
- The TBB library has integrated, as a standalone utility not necessarily linked to the TBB programming environment, a partial implementation of the C++11 standard dealing with thread management and synchronization: the services proposed in the <thread> and <condition_variable> headers.

One should also be aware that both Boost and TBB propose very useful extended facilities that have not been incorporated into the C++11 standard, like the reader-writer locks discussed in Chapter 8.

3.6.2 C++11 ERROR REPORTING

C++11 thread library functions do not return error codes. Return values, if any, only return relevant information resulting from the function operation. As a C++ library, C++11 threads throw exceptions in the case of failure. Remember that exceptions can be looked at as non-local return values, not aimed only to the direct caller. When an exception is thrown, there is a normal function returned to the caller. But if the caller has not explicitly set up a try-catch block to catch it, the exception is returned to its caller, and so on and so forth. The exception keeps climbing the hierarchy of nested function calls until somebody has set up the try-catch safety net that catches it. In the absence of any safety net, the exception writes a message when it arrives at the main function, and terminates the program.

The C++11 standard is very explicit on the exceptions thrown by the different library functions. Catching exceptions is often employed by library writers, but in the applications programming we will be developing in this book there is hardly the need of explicitly catching them. After all, even if we do catch them, what else can we do other than abort the program?

3.6.3 C++11 DATA TYPES

In C++11, the new data types for multithreading are, naturally, C++ classes defined in the std namespace. Library functions are most often—but not always—*public member functions* of these classes. We will soon discuss how they operate.

Rather than explicitly passing the address of an object, as is the case in Pthreads and Windows, the standard C++ semantics is employed: member functions are called on an object (whose address, remember, is implicitly passed to the function by the C++ compiler, via the this pointer).

3.6.4 CREATING AND RUNNING C++11 THREADS

C++11 thread creation and management is implemented in the std::thread class, representing a thread. Its role is very clear: *when an object instance of this class is created, a new thread is launched*. This std::thread object can be seen as representing the new thread's identity, in the same way a pthread_t object represents a thread identity in Pthreads.

A new thread is therefore started by constructing an instance of std::thread and passing a pointer to a thread function in the constructor of the object. The C++ standard is less strict than Pthreads/Windows in what concerns the signature of the thread function: the thread function can be any function that returns void, with an arbitrary number of arbitrary arguments. Listing 3.12 shows the C++11 protocol for thread creation:

```
void thread_fct(arg1, arg2, ...);        // definition of thread function
...
std::thread T(thread_fct, arg1, arg2);   // launch the thread T
```

LISTING 3.12

Creating a C++11 thread

Several important points are worth noting:

- The new thread is created and starts running as soon as the std::thread object T is created. The new thread immediately starts executing the thread function passed as argument.
- As stated above, *the thread function can take an arbitrary number of arguments of practically any type*. In fact, it can take arguments of any *copyable* type, namely, a type that disposes of a copy constructor.
- The thread constructor exhibited in Listing 3.8 above *copies* the thread function address and arguments to some internal working place, where they will be read once the thread environment is created and the thread function is called. This is fine, as long as the thread function arguments are passed by value. However, it may happen that the thread function expects *a reference* to some external variable to return a result. In this case, special care is needed. This issue is discussed in detail in the examples that follow.
- The C++ library *does not* directly manage return values from the thread function as Pthreads/Windows do. The function used to launch a new thread always returns void. However, C++11 proposes an independent mechanism, the future facility, that allows a thread to recover asynchronous data values generated by another thread.

Remember also that in C++ it is possible insert a function object—namely, an object that acts also as a function—anywhere a pointer to a function is expected. Function objects are a powerful C++ feature. They are heavily used in the STL, or in other multithreading libraries like TBB and Boost. For this reason, a discussion of their properties is proposed in Annex B, in particular explaining how function objects can replace pointers to functions, with extended capabilities.

In a function object, *the name of the object can be used as a function name*, and the member function called in this way is the operator() member function, whose signature—arguments and return values—can be adapted to the application requirements. Here is the way a function object is defined and passed to the std::thread constructor:

```
class mytask      // define function object class "task"
   {
   // private data
   // private member functions

   public:
    ... // Constructor, destructor
    ... // other member functions

    void operator()(arg1, arg2, ...)  // this makes task a function object
       {
       // code for operator(...) function
       }
   ...
   };

...
mytask tk;                             // define function object tk
std::thread T(tk, arg1, arg2, ...):   // launch thread
```

LISTING 3.13

Creating a C++11 thread with a function object

Notice the flexibility of this approach. The function object constructor can also be used to pass data values that may be required for the operation of the member functions. We will see this in detail in one of the examples that follow.

Comments on threads as objects

The fact that C++11 threads are associated with objects may initially induce some confusion. Normally, C++ objects are seen as black boxes that encapsulate data and provide services to client code via their public member functions. Threads, on the other hand, are just asynchronous execution streams. So how are these two views linked together?

We know that the std::thread object T represents the identity of the new execution stream launched when the object is created (Figure 3.1). However, as an object, the thread T is an instance of a class, with a number of member functions that can be called by client code. Therefore, the natural question that comes up is: where do the client codes requesting the services of T come from? The answer is that *they come from other threads in the application*. Indeed, other threads can access the T member functions to produce some action on the thread it represents. Notice that when this happen the called member function acting on T runs in the caller thread environment, not in the target thread represented by T.

The C++11 thread termination interfaces implement the same concepts as Pthreads/Windows: threads can be joined or detached. There are, however, a few subtleties that programmers should be aware of. In C++11, join() and detach() are member functions of the std::thread class, corresponding to operations that need to be performed on a thread.

- Any created thread must be either joined or detached *before the thread object that represents the thread is destroyed*. If that is not the case, the whole process terminates when the thread object is destroyed.
- A thread that wants to join another thread T calls T.join(). When this happens, the caller thread moves to a blocked state and waits, until the execution stream associated to T terminates. After a thread is joined, its execution stream no longer exists. The associated T object represents, until it is destroyed, a special object called not-a-thread.
- On the other hand, a thread can be detached by calling T.detach(), to let the thread represented by T fade away and disappear gracefully after termination. After detachment, its execution stream continues naturally to execute until the thread function terminates, but the associated object T becomes invalid and, again, it represents a not-a-thread object. After detachment, a thread can no longer be acted upon by other threads.

```
...
void task_fct(...);
std::thread T(task_fct):   // launch new thread T
...
T.join();           // caller thread waits for T to terminate
...
T.detach();         // Function returns immediately,
                    // T is no longer related to the detached thread
```

LISTING 3.14

Joining or detaching threads

The stack size.

The C++11 standard does not propose any explicit way of modifying the default thread stack size. Boost has a special class attrib whose only role is to fix the stack size of the caller thread.

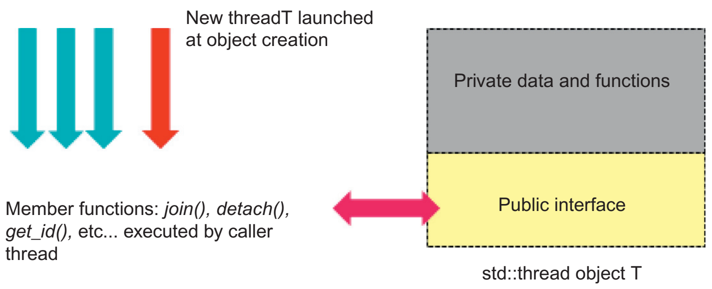

New threadT launched
at object creation
Private data and functions
Public interface
Member functions: join(), detach(), get_id(), etc... executed by caller thread
std::thread object T

FIGURE 3.1

std::thread object T.

Scope of C++11 thread objects

An important issue is where and how std::thread objects should be created: as global objects or as objects local to the thread that creates the new thread? The scope of the thread objects depends on the way they will be accessed. A thread object can be created as a local object to the parent thread, instantiated at the appropriate place where the new thread needs to start running. However, in this case, only the parent thread will be able to join or detach it. This may be sufficient in many applications.

If a more general context is needed in which any running thread needs to access other thread objects, they should be declared with global scope. But in this case their constructor will be called before main() starts, and we may end up with new threads running too early, before the main() function has set up the stage of the application. In order to avoid this, a *lazy initialization* strategy is required, by declaring *global pointers* to std::thread objects. Later on, main() or any other thread will create the new, global thread objects at the right place with a call to the new operator that initializes the global pointers. The next section provides one explicit example of this approach.

Useful member functions of std::thread

The C++11 documentation can be consulted for a complete description of the std::thread class. Here, we quote two member functions frequently used in the applications, in particular in the vath library. In the functions and data names below, the std:: namespace is understood.

- thread::sleep_for(duration d). This function call puts the target thread in a blocked state for time interval d. It is useful to simulate some amount of work done, by slowing down for a given elapsed time interval the thread progression. In the function signature given above, *duration* is a type defined in the <chrono> header, and the Annex at the end of this chapter describes the way this data type operates. The examples in the next section show how the sleep_for() member function is used in practice.
- thread::thread_id thread::get_id(). C++11 defines a data item thread_id for each thread object, and the function get_id() returns its value. Now, since the thread object itself identifies a thread, why do we need another thread identifier? The point is that very often thread identities need to be compared—we have seen an example in the GetRank() functions in Pthreads and Windows—and C++11 overloads the comparison operators ==, != for thread_id objects. C++11 also overloads the output operator << for a thread_id object, so the thread identities can easily be printed.

std::this_thread global functions

As explained before, member functions run in the caller thread environment, and act on the target thread object on which they are called. If main() calls T.sleep(d), then main puts T to sleep. But what happens if we want an action to take place *in the caller thread environment*? For example, how can a thread put itself to sleep for a given elapsed time interval?

To handle this issue, C++11 defines some global, non-member functions in the std::this_thread namespace. Here are the non-member function versions of the two member functions discussed before (the std namespace is always understood):

- this_thread::sleep_for(duration d). This function call puts the caller thread in a blocked state for time interval d.
- thread::thread_id this_thread::get_id(). Returns the thread_id of the caller thread.

3.6.5 EXAMPLES OF C++11 THREAD MANAGEMENT

A few examples of C++11 thread operation are presented next, to illustrate and complete the previously introduced concepts.

Launching worker threads

Example 8. In the first example, a worker thread waits in a blocked state for a given number of seconds to simulate a long computation or I/O operation. The main thread launches the worker thread and waits for its termination by calling its join() member function.

The thread function executed by the worker thread is given in Listing 3.15. For clarity, the initial and final messages the worker thread sends to stdout have been suppressed, and only the real code is kept: two lines of code that put the thread to sleep for a number of milliseconds passed as argument. The first line creates a milliseconds duration object defined in the <chronos> header. In this way, the mSecs integer passed as argument is interpreted as a number of milliseconds. For a duration coded in seconds, a std::chrono::seconds object should be constructed. The second line passes this duration object to the sleep_for() function in the this_thread namespace, as discussed in the previous section. The thread then sleeps for the duration is passed as an argument.

The main() function constructs the std::thread object T by passing to the constructor the address of the thread function (which is its function name) and the value of the integer argument to be passed to it. In our case, the worker thread waits for 3 s. Then, main() joins the worker thread, waiting for its termination.

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

void ThreadFunc(unsigned mSecs)
   {
   std::chrono::milliseconds workTime(mSecs);  // create duration
   std::this_thread::sleep_for(workTime);      // sleep
   }


int main(int argc, char* argv[])
   {
   std::thread T(ThreadFunc, 3000);    // thread created
   std::cout << "main: waiting for thread " << std::endl;
   T.join();
   std::cout << "main: done" << std::endl;
   return 0;
   }
```

LISTING 3.15

Launching a worker thread

Example 8: Cpp1_S.C

To compile, run make cpp1s.

Example 9. This example modifies the way the thread in the previous example is created. The thread function is always the same. In Example 8, the std::thread object T is created as a local object inside the main() function, in the main thread stack. Consequently, this thread object can only be accessed and acted upon by the main thread.

To create a thread with global scope, the *lazy initialization* strategy discussed above is adopted: rather than declaring a global thread object, a *global pointer* to a thread object is declared. This global pointer is, when the time comes, initialized by main() by a call to the new operator, and it is at this point that the new thread starts running because it is at this point that its constructor is called. This approach is shown in Listing 3.16.

```
std::thread *Wth;     // global data

int main(int argc, char* argv[])
   {
   std::cout << "main: startup" << std::endl;
   Wth = new std::thread(ThreadFunc, 3000);  // thread created
   std::cout << "main: waiting for thread " << std::endl;
   Wth->join();
   std::cout << "main: done" << std::endl;
   delete Wth;
   return 0;
   }
```

LISTING 3.16

Launching a worker thread

Notice that, after the join() function returns, Wth is still a valid pointer, pointing to a not-a-thread object still sitting in the heap. Deleting this pointer at the end of main() restores the memory allocated in the heap.

Example 9: Cpp2_S.C

To compile, run make cpp2s.

Example 10. Next, a thread is created using a function object. Listing 3.17 defines a class used to compute the square root of a number using the iterative Newton's method. This is an ordinary class, with an operator() member function. This class requires three pieces of information: the target number whose square root is to be computed, a guess for the square root value used a starting point of the iterations, and an upper limit to the accepted number of iterations in the calculation.

To demonstrate the flexibility of the function object approach, this information is passed in two different ways: the target value and the initial guess are passed as arguments to the operator() function, but the upper limit to the number of iterations is passed via the function object constructor, as shown in Listing 3.17.

```
class Worker        // This is the function object
   {
   private:
    unsigned Niters;

   public:
    Worker(unsigned N) : Niters(N) {}

    void operator()(double target, double guess)
       {
       std::cout << "Worker: computing sqrt(" << target
                 << "), iterations = " << Niters << std::endl;

       // use Newton's method
       // -----------------
       double x;
       double x1 = guess;
       for(unsigned i=0; i<Niters; i++)
          {
          x = x1 - (x1*x1 - target) / (2 * x1);
          if(fabs(x-x1) < 0.0000001) break;
          x1 = x;
          std::cout << "Iter " << i << " = " << x << std::endl;
          }
       std::cout << "Worker: answer = " << x << std::endl;
       }
   };

int main(int argc, char* argv[])
   {
   Worker w(612);              // object w is constructed
   std::thread T(w, 25, 4);    // thread function is w(25, 4)

   std::cout << "main: waiting for thread " << std::endl;
   T.join();
   std::cout << "main: done" << std::endl;
   return 0;
   }
```

LISTING 3.17

Using a function object

The main() function code exhibits the operation of the function object used to create the thread. The thread function the new thread executes is the operator() member function of the Worker class. First, a Worker object w is created (the name is arbitrary). Once the object is created, the operator() member function can be called by using the object name; namely, by invoking w(25, 4) to compute the square root of 25 starting from a guess of 4. The thread constructor is therefore invoked, as before, by passing as arguments the function object name and the arguments to be passed to the operator() function.

Example 10: Cpp3_S.C

To compile, run make cpp3s.

Notice that in this example the result of the computation is printed to the screen, but it is not returned to the main thread. Given the flexibility of function objects, we could imagine returning the result to the main thread by defining an additional public double result field in the Worker class, where the final result is stored before the operator() function completes. Then, the main thread would read the result from the function object w after the worker thread is joined.

The idea is good, but it does not work as such, because of the *copy semantics* used by the thread constructor mentioned before: the function object arguments are passed by value, and the function object itself is copied to another object used internally by the library. Therefore, the return value will not be stored in the original function object w known to main(), but in the internal working copy used by the library. To make this idea work, *the object w known by main() must be passed by reference*, in the way we discuss next.

Passing values by reference

Example 11. Consider first a simple example: a thread function whose only purpose is to update an external variable passed by reference, as shown in Listing 3.18. The thread function adds 10 to the integer passed by reference.

```
void ThreadFunc(int& N)  // a simple thread function
   { N += 10; }

int main(int argc, char* argv[])
   {
   int N = 10;
   std::cout << "\n Initial value of N = " << N << std::endl;
   // ---------------------------------
   std::thread T1(ThreadFunc, N);
   T1.join();
   std::cout << "First value of N = " << N << std::endl;
   // ---------------------------------
   std::thread T2(ThreadFunc, std::ref(N));
   T2.join();
   std::cout << "Second value of N = " << N << std::endl;
   // ---------------------------------
```

```
    return 0;
    }
```

LISTING 3.18

Passing arguments by reference

The main function launches successively two threads. In the first thread T1, the target integer N is passed by value to the constructor, and it is a reference to this value internally stored by the runtime system that will be passed to the thread function. The update will take place, but its target is the internal copy of N, and not the value that main() knows. The program execution shows that our known value of N is not updated.

Next, a second thread T2 is launched, but now N is passed to the constructor using the std::ref(N) qualifier. The program execution shows that now N is correctly updated.

Example 11: Cpp4_S.C

To compile, run make cpp4s.

Example 12. Next, Example 10 is reconsidered, in order to show how to implement the previous idea for returning the result of the computation to the main thread. The Worker function object class is modified by adding a public field double result, where the operator() function stores the result. Then, the std::ref(w) qualifier is used in the constructor to pass the function object itself by reference. In this way, it is the function object known to us and not an internal copy that holds the result at the end of the computation.

```
class Worker
   {
   private:
    unsigned Niters;

   public:
    double result;
    ...
    // the rest is the same
   };

int main(int argc, char* argv[])
   {
   Worker w(500);
   std::thread T(std::ref(w), 25, 7);  // w passed by reference
   T.join();
   std::cout << "main: result obtained is " << w.result << std::endl;
   return 0;
   }
```

LISTING 3.19

Passing a function object by reference

Example 12: Cpp5_S.C

To compile, run make cpp5s.

Transferring thread ownership

It is clearly understood that a std::thread T object represents a thread, which is a unique resource. If a thread object could be copied, we could end up with several objects representing the same unique resource, which is not acceptable. Therefore, *std::thread objects are not copyable*. The copy constructor and the assignment operator are by design deleted in the std::thread class. Threads, however, can be moved from one thread object to another. A thread linked to a thread object T1 can be transferred to another empty object T2, which becomes its identifier, and the initial object T1 becomes a not-a-thread after the move.

Example 13. Listing 3.20 shows how this ownership transfer is performed. A thread T1 that sleeps for 5 s is run. The whole point is to have the thread active long enough so that the transfer operation can be safely performed. When the thread T1 is launched and its thread_id is immediately printed, the code execution reports a positive integer value.

Next, a new thread object T2 is created calling std::move(T1), and the new T1 identity is printed. The reported value is 0, the code for not-a-thread. On the other hand, the reported identity value for T2 is the same as the original T1 identity.

```
void ThreadFunc()
   {
   std::chrono::seconds workTime(5);
   std::this_thread::sleep_for(workTime);  // sleep
   }

int main(int argc, char* argv[])
   {
   std::thread T1(ThreadFunc);       // launch thread  T1
   std::cout << "T1 identity = " << T1.get_id() << std::endl;
   std::thread T2 = std::move(T1);  // transfers thread ownership
   std::cout << "New T1 identity = " << T1.get_id() << std::endl;
   std::cout << "T2 identity     = " << T2.get_id() << std::endl;
   T2.join();
   return 0;
   }
```

LISTING 3.20

Transferring thread ownership

We will come back to the ownership transfer issue in the last example in this section.

Example 13: ThMove_S.C

To compile, run make thmoves.

Launching a worker team

Example 14. This example is the CPP11 version of the Hello4_P.C Pthreads example in which a set of N threads is created. Each worker thread will be identified by an integer rank in [1, N]. Worker threads print a hello message, indicating their rank. The interesting part of this example is the C++11 version of the GetRank() function previously discussed, as well as the procedure adopted for the allocation of N worker threads in the heap.

```
std::thread   **Wth;    // array of thread pointers
int           nTh;      // number of threads

int GetRank()           // auxiliary function
   {
   std::thread::id my_id, target_id;
   int n, my_rank;

   my_id = std::this_thread::get_id();
   n = 0;
   do
      {
      n++;
      target_id = Wth[n]->get_id();
      } while(my_id != target_id && n < nTh);

   if(n<=nTh) my_rank = n;  // if rank OK, return
   else my_rank = (-1);     // else, return error
   return my_rank;
   }


void helloFunc()            // thread function
   {
   std::chrono::milliseconds delay1(500);
   std::this_thread::sleep_for(delay1);   // first delay
   int rank = GetRank();
   // - - - - - - - - - - - - - - - - --
   std::chrono::seconds delay2(rank);
   std::this_thread::sleep_for(delay2);   // second delay
   std::cout << "Hello from thread number " << rank
             << std::endl;
   }

int main(int argc, char* argv[])
{

   if(argc==2) nTh = atoi(argv[1]);
```

Continued

```
else nTh = 4;

Wth   = new std::thread*[nTh+1];     // allocate thread array
for(int n=1; n<=nTh; ++n)            // create threads
   Wth[n] = new std::thread(helloFunc);

for(int n=1; n<=nTh; ++n) Wth[n]->join(); // Join threads
std::cout << "main: done" << std::endl;

for(int n=1; n<=nTh; ++n) delete Wth[n];  // delete threads
delete [] Wth;                            // delete array
return 0;
}
```

LISTING 3.21

RunTeam1_S.C

An array Wth[n], n=1, ... N of std::thread* pointers is allocated. Then, each one of these pointers is initialized with a new std::thread object. The global object Wth declared outside main() is a pointer to an array of std::thread objects. This array has size (N+1), because we want to manipulate an *offset 1* vector of thread identities. Therefore, the first slot Wth[0] is not used, and thread objects are allocated starting from Wth[1]. The main function launches and then joins the worker threads.

A few comments about the thread function are in order. When created, the worker threads are first delayed for 500 ms. The reason is that they all call GetRank() to know their rank, and that this function scans the global array Wth. Therefore, the purpose of this initial delay is to ensure the array is completely initialized when accessed. Then, there is a second delay in which each thread sleeps for an interval in seconds equal to their rank. The purpose of this delay is to separate the messages written to stdout.

Example 14: RunTeam1_S.C

To compile, run make runt1s.

Example 15. Finally, this last example shows how to do the same thing (allocating and running a set of nTh worker threads) in a way that shows again the ownership transfer in operation. We will launch nTh threads and stock them in a STL vector container of threads.

The thread function is very simple: it gets a rank integer as argument, sleeps for a duration equal to its rank in seconds, and prints a message to stdout, reporting its rank.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <vector>

void helloFunc(unsigned rank)
```

```
   {
   std::chrono::seconds workTime(rank);
   std::this_thread::sleep_for(workTime);  // sleep
   std::cout << "Hello from thread number " << rank
             << std::endl;
   }

int main(int argc, char* argv[])
   {
   unsigned n, nTh;
   if(argc==2) nTh = atoi(argv[1]);
   else nTh = 4;

   std::vector<std::thread> workers;
   for (n=0; n<nTh; ++n)
       workers.push_back(std::thread(helloFunc, n));

   for(auto &th : workers) th.join();
   std::cout << "Main: done" << std::endl;
   return 0;
   }
```

LISTING 3.22

RunTeam2_S.C

Let us now look at the main function, where there is a lot of action going on in a couple of lines. First of all, a STL vector container of std:: thread objects, called workers, is declared. It is initially an empty vector. Next, threads are created and inserted in this container. Notice that the explicitly created threads are temporary objects that are bound to be destroyed at the end of the loop iteration. These temporary objects are inserted into the vector container, but we know they cannot be copied. Rather, they are moved, which means the thread ownership is transferred to the thread object created inside the container. At this point, the initial temporary thread object becomes a not-a-thread object, and can safely be destroyed at the end of the loop iteration.

Example 15: RunTeam2_S.C

To compile, run make runt2s.

3.7 SPool UTILITY

Several of the previous examples implemented a *fork-join* pattern in order to perform a given parallel treatment. A team of worker threads was created with an explicit assignment coded in the thread function, and then the worker threads were joined by the client thread. In our simple previous examples, all the worker threads were doing the same job (printing a message to *stdout*). A very common pattern in more realistic parallel applications occurs when the worker threads are asked to performing the same action on different sub-sectors of the data set. This very common work sharing construct is called a

SPMD (Single Program, Multiple Data) parallel pattern. It only requires the forking and joining of a set of worker threads, all executing the same thread function on different sectors of the data set.

The SPool utility, included in the vath library, provides a simple interface for implementing a fork-join parallel pattern. As all the classes in this library, SPool is portable in the sense that it has a Pthreads, Windows, and C++11 implementation. This C++ class encapsulates the previously discussed basic libraries thread management interfaces, and it also provides a few useful additional utilities. Its public interfaces are very close in spirit to the way OpenMP implements a fork-join parallel pattern (to be discussed later in this chapter). However, as we will see, there are some differences in design and programming style.

We will henceforth speak of *creating a parallel region* when forking and joining a set of worker threads to perform a parallel treatment. In all the examples that follow, this happens only once in the application, and one can then imagine creating the new worker threads if and when they are needed, and joining them when they terminate and disappear from the scene. However, in many real applications like the ones we will meet in later chapters the creation of a parallel region occurs more than once, often inside a loop that is executed a huge number of times. When this happens, creating new threads every time they are needed and destroying them when they finish introduces too much thread management overhead. Remember that, among other things, every time a new thread is created or destroyed, the stack memory buffer that goes with it must be allocated or released.

In this case, it is more efficient to create the worker team only once, and put them in a blocked state (as indicated in Chapter 2) until some other thread needs their services to run a parallel region. This is efficient because, when sleeping in a blocked state, the worker threads do not use CPU resources and therefore they do not compete for resources with the remaining threads in the application. This is the strategy the SPool adopts, as well as any other high-level management utility like OpenMP or TBB. Later on, we will give some hints on the way this software architecture is implemented

A set of threads available for usage in the way described above is called a *thread pool*. The SPool utility is in fact a thread pool limited to the execution of SMPD parallel jobs, which explains its name.

Role of the client thread

We call the *client thread* the thread that creates the parallel region. Typically, the client thread is the main thread that starts with the process, but this is not required. Thread environments are democratic, and any thread can launch a new worker thread team. Our main design choice is that the client thread *does not join the workers team, and does not participate in the execution of the parallel job*. After activating the workers team, the master thread may continue to do something else and, at some point, it waits for the termination of the activity of the worker threads.

This is slightly different from OpenMP. We will soon see that, in OpenMP, a master thread that reaches a parallel section and launches a team of worker threads suspends whatever task is executing and becomes a *de facto* member of the worker team. The master thread resumes execution of the suspended initial task when the parallel treatment is finished. It would look as if this design choice enables a better usage of CPU resources, if the parallel section engages as many threads as available cores. However, in our case, a master thread waiting to join other threads performs an idle wait and moves to a blocked state *not using CPU resources*. Therefore, if the master thread has nothing to do, all the executing cores are fully available to the worker threads. To sum up: our design choice adds some flexibility to the parallel code because:

- It allows a client thread to continue to do some useful concurrent work after launching the worker team. But if there is no useful work available, CPU resources are not wasted.
- It allows a master thread to create several totally independent worker teams, and execute several independent parallel routines in parallel. This parallel pattern, not easy to implement in OpenMP, will be discussed in detail in Chapter 12.

Here is the public interface of the SPool class:

THE SPOOL PUBLIC INTERFACE

1—Constructing a Spool Object

`SPool(int nTh, double stksize=0.0)`

- Constructor of the team of nTh worker threads.
- Second argument is, in Pthreads, a scale factor that multiplies the default stack size.
- Second argument is, in Windows, a new value for the stack size.
- If the second argument is zero (default), the stack size is not modified.
- Called by client (master) threads.

2—Forking and joining threads

`void Dispatch(void (*fct)(void *), void *arg`

- Wakes up and activates the worker threads, executing the task function passed as argument
- Returns immediately after activating the worker thread set.
- Worker threads are identified by an integer rank in [1, N].
- The rank can be used to select the work to be done by each thread.
- Called by client threads.

`void WaitForIdle()`

- The caller thread moves to a blocked state until all the worker threads complete their previous assignment.
- Called by client threads.

3—Utilities, called by worker threads

`int GetRank()`

- Returns the rank of the caller worker thread.

`void ThreadRange(int& beg, int& end)`

- Work sharing utility function.
- Distributes indices in an global integer range across worker threads.
- beg and end are input-output parameters.
- On input, [beg, end) is the global index range.
- On output, [beg, end) is the index sub-range allocated to the caller thread.

`void CancelTeam()`

- Called by a worker thread to cancel and stop the activity of all the worker team.
- The caller threads informs all other workers that they must terminate.
- This call also terminates the ongoing job.

`void SetCancellationPoint()`

- Called by a worker thread, to check for cancellation request.
- If cancellation is requested, the current task function terminates.

This public interface is simple: it involves the constructor and destructor of the thread set, as well as member functions to launch a parallel job and wait for its termination. There are also three other utilities useful for some specific applications. Each worker thread in the set is uniquely identified by an integer value called rank in the range $[1, N]$. Any worker thread in the team can learn about its rank by calling the member function GetRank().

Creating SPool objects

The constructor of a SPool object takes two arguments:

- The number N of threads in the set.
- The second argument—a double—corresponds in the Pthreads implementation to a scale factor for the stack size: the default stack size provided by the operating system is multiplied by this scale factor. In the Windows implementation, this value, recast as a long, becomes the new stack size that replaces the default size. In the C++11 implementation, this argument is ignored. If the scale factor passed to the constructor is 0 (the default value), the default stack size is not modified.

These different ways of handling the stack size (something that in any case is rarely needed) are motivated from the fact that in the Pthreads implementation it is possible to inquire about the current stack size value and multiply it by the scale factor. In other environments, instead, there is no direct way of knowing the current stack size value. Therefore, one needs to try a new, absolute value of the stack size and pass it to the constructor.

Running the thread team

The Dispatch() member function activates the N worker to execute the function passed as argument. This function returns immediately after activating the N worker threads. As discussed above, the client thread that calls Dispatch() has the choice of continuing to do some other useful work in parallel with the work done by the thread team, before waiting to join the worker threads.

The master thread at some point calls the member function WaitForIdle(). This function operates *as if* the master thread waited for the termination and joins one by one all the worker threads. Listing 3.23 shows how the SPool objects are used in practice

Creating and using SPool objects

A workers team is created by constructing a SPool object, and the point is: where and how should this object be constructed?

If all we plan to do is fork-join the workers team with the Dispatch–WaitForIdle function calls, then only the client thread needs to access the SPool object, and in this case there is no harm in declaring it as a local object to the client (master) thread. However, most of the time the worker threads need to be able to access the SPool object in order to call some of the utility functions. In this case, the SPool object must be declared with global scope (outside main()), so that it can be seen and accessed by all the threads in the application. From now on, we will most of the time declare SPool objects with global scope.

There are still two possible options to create this global SPool object. If the number of worker threads is hardwired in the application and known at compile time, then it is possible to directly create a global object, initialized by the runtime system before main() starts. Notice that in this utility there is no problem in creating the worker team too early, for reasons that will soon be clear. If, instead, the number of threads is obtained from the command line or from an input file, we are lacking information to create the object in this way. In this case, *a global pointer* to a SPool object is declared. The pointer is initialized later on by main() with a call to the new operator. Listing 3.23 shows how this option proceeds.

```
void th_fct(void *P);
SPool *TS;             // SPool pointer

int main (int argc, char *argv[])
   {
   int nTh;                // read from command line
   TS = new SPool(nTh);    // default values for 2nd arg
   TS->Dispatch(thfct, NULL);
   // - - - - - - - - - - - - - - -
   // Here, main can do other work
   // - - - - - - - - - - - - - - -
   TS->WaitForIdle();             // main joins all worker threads
   delete TS;
   }
```

LISTING 3.23

Using SPool objects

ThreadRange() utility

This function is called by worker threads in a work sharing context where the purpose is to parallelize a loop. In this case, a set of contiguous vector indices must be distributed among the worker threads, to select the sectors of the data set on which each thread will act. This function determines the optimal sub-range of indices for a given worker thread, resulting from a fair work distribution among workers. This is possible because each thread knows the number of threads in the team, as well as its own rank value and the global index range, so that it can compute the optimal sub-range of indices it must handle. See the details of the operation of this function in the example that follows dealing with the addition of two vectors.

CancelTeam() utility

This is the only sophisticated piece of programming in this class. A worker thread calling this function forces the immediate cancellation of the ongoing parallel job. There are some mild implementation differences:

- In Pthreads, this utility relies on a very efficient, native thread cancellation service. The worker threads terminate their lifetime. After cancellation, the thread pool is no longer active, and cannot be reused for another parallel job. But it is always possible to construct on the fly another pool if needed.

- In Windows and C++11, this utility relies on an advanced synchronization tools described in Chapters 8 and 9. In this case, *the parallel job is cancelled*, and the worker threads remain active to perform another job if needed.

3.7.1 UNDER THE SPOOL HOOD

It is instructive to understand the way the SPool objects operate, by creating a pool of threads and putting it in a blocked state waiting to be activated by client threads. Figure 3.2 shows the pool operation:

- Threads are created and launched by the SPool constructor. They execute a thread function defined internally in the class, which implements the behavior that follows.
- As soon as the threads are created, they go to sleep in a blocked state.
- A client thread that dispatches a job initializes an internal pointer to the function to be executed and releases the blocked threads.
- At wake up, the worker threads call the task function via the internal function pointer, and goes back to a blocked state when they complete its execution.
- And so on, and so forth. The worker threads keep executing this infinite loop as new job requests are dispatched.
- When the SPool destructor is called, the worker threads are released, they exit the infinite loop, and are joined by the external master thread that called the destructor.

Notice that the task function passed to the Dispatch() member function is not a thread function in the usual sense, it is a function called by the already existing threads. For this reason, this function does not need to adopt the imposed thread function signature in Pthreads, and it returns void instead of void*.

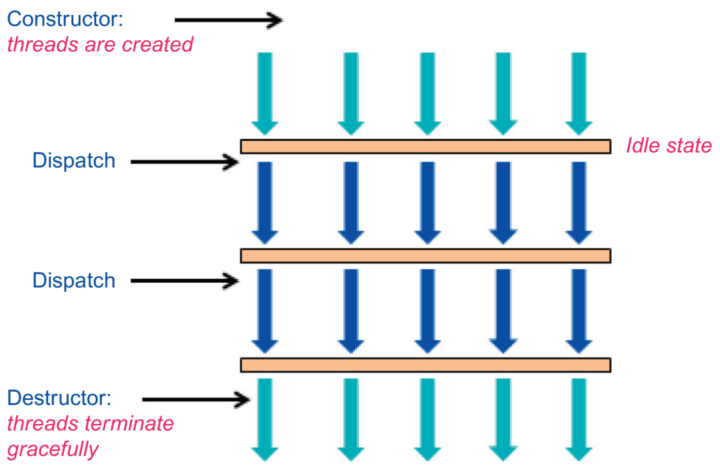


FIGURE 3.2

Operation of the SPool thread worker team.

The source code for the SPool class is simple and compact because it relies on a high-level synchronization utility—a blocking barrier—discussed in Chapter 9. An example is included in Chapter 9 that illustrates the inner behavior of the SPool class.

3.8 SPool EXAMPLES

Let us now take a look at a few simple examples based on the SPool utility, illustrating some common issues in applications programming. These examples use some general-purpose utility classes from the vath library. Their usage is simple and evident:

- **Class Rand**. Objects of this class generate uniformly distributed random doubles in [0, 1].
  - An object R is declared as Rand R(int n), where n is a seed that initializes the generator.
  - Random doubles d are obtained by d = R.draw().
- **Class CpuTimer**. Objects of this class are used to measure execution times of selected code blocks contained between calls to the member functions Start() and Stop().
  - Execution times are reported by a call to Report().
  - This function prints both the *user time*, which is the total execution time of all the threads in the application (equal to the sequential execution time), and the *wall time*, which is the real time elapsed.
  - Obviously, the parallel speedup is the ratio of user to wall time. On a dual-core laptop, for example, the optimal performance is a wall time equal to half the user time.

Given the simplicity of the examples, there is no point in using too many threads to exhibit the speedup resulting from the parallel computation. Two worker threads are often hard wired in the source codes.

3.8.1 ADDITION OF TWO LONG VECTORS

This is a very simple *embarrassingly parallel* problem. The initial sequential source code is in file add_vectors.C. The relevant part of the sequential code is listed below:

```
double A[VECSIZE], B[VECSIZE], C[VECSIZE];

int main (int argc, char *argv[])
   {
   int i, j, nsamples;

   // Get nsamples from the command line
   // ------------------
   for(int j=0; j<nsamples; n++)
      {
      for(n=0; n<VECSIZE; n++) C[n} = A[n] + B[n];
      }
   ...
   }
```

LISTING 3.24

Adding two vectors : add_vectors.C

The vector addition is repeated nsamples times to be able to measure reasonable execution times. The parallel code is in file AddVectors.C. The main thread launches two worker threads, and each one computes the sum of the first or the second half of the vectors. This application is qualified as *embarrassingly parallel* because no communication whatsoever is required between the worker threads or between the worker threads and the main thread.

Details of the thread function are shown below, because it uses the ThreadRange() work sharing utility. This function receives *a reference to two integers* called beg and end in Listing 3.25, which are *input-output parameters*. On input, these integers are initialized with the values of the global vector index range. This member function knows the rank of the caller thread as well as the number of threads. Therefore, it can compute the index range appropriate to the caller thread. When the function returns, the function arguments contain the caller thread index sub-range.

Notice that we have adopted in this book the C++ STL conventions for index ranges. They are defined by the first and the one past the end values, so ranges are described by the half open interval $[beg, end)$.

```
SPool TH(2);        // two worker threads.

void *thread_fct(void *P)
   {
   int beg = 0;                // initialize [beg, end) to global range
   int end = VECSIZE;
   TH.ThreadRange(beg, end);   // now, [beg, end) is the index range for
                               // this thread
   for(int j=0; j<nsamples; j++)   // compute nsample tiles the sum
      {
      for(int n=beg; n<end; n++) C[n] = A[n] + B[n];
      }
   }
```

LISTING 3.25

Thread function in AddVectors.C

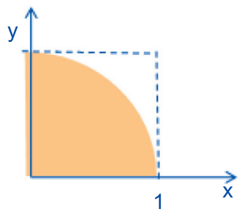
Executing the sequential and the parallel codes for a very large number of samples shows without surprise a speedup of 2.

Example 16: AddVectors.C

To compile, run make addvec. The number of threads is 2. The number of samples is passed from the command line (the default is 1000000).

3.8.2 MONTE CARLO COMPUTATION OF π

The algorithm for the Monte Carlo computation of π is very simple. Look at the Figure 3.3, where a box of side 1 is shown, containing a quarter of a disk of radius 1. The ration of the area of the disk to the area of the box is, naturally, $\pi/4$.



```
double ComputePi()
   {
   double x, y, retval
   long n, count=0;

   for(n=0; n<SAMPLE; n++)
       {
        x = R.draw();
        y = R.draw();
        if( (x*x+y*y) <= 1.0) count++;
        }
    retval = 4.0 * count / SAMPLE;
    return retval;
    }

R provides uniform random numbers
in [0, 1].
```

FIGURE 3.3

Algorithm for the Monte Carlo computation of π.

Suppose that a number SAMPLE of random points of coordinates (x, y) are generated inside the box, by using for x and y uniformly distributed random numbers in [0, 1]. These points will uniformly fill the box. Let count be the number of these random points that fall inside the disk. Obviously, the ratio count/SAMPLE will approach the ratio of areas—i.e., $\pi/4$—when SAMPLE is very large. This ratio therefore determines π. The figure also shows the code of a function that computes π.

A parallel version of this code can be written as follows:

- The main thread launches two worker threads. They will share the computational work by executing the code indicated above using SAMPLE events per thread.
- Each worker thread determines its own number count1 or count2 of accepted events. Then, they communicate this number to the main thread, which puts everything together and computes the final value of π as $\pi = 2.0(count1 + count2)/SAMPLE$.

The sequential code for this problem is in source file calcpi.C. Here is the parallel code, contained in the source file CalcPi.C:

```
#include <stdlib.h>
#include <CpuTimer.h>
#include <iostream>
```

Continued

```
#include <Rand.h>
#include <SPool.h>

using namespace std;

SPool TH(2);                    // two worker threads
unsigned long count[3];         // storage of partial results
long nsamples;                  // initialized by command line
Rand R(999);                    // random number generator

void task_fct(void *P)
   {
   unsigned long ct;
   double x, y;
   int rank = TH.GetRank();     // get thread rank in [1, nTh]
   ct = 0;
   for(int n=0; n<nsamples; n++)
      {
      x = R->draw();
      y = R->draw();
      if((x*x+y*y) <= 1.0 ) ct++;
      }
   count[rank] = ct;            // store partial result in global array
   }


int main(int argc, char **argv)
   {
   CpuTimer TR;
   double x, y, pi;

   if(argc==2) nsamples = atoi(argv[1]);
   else nsamples = 1000000;

   TR.Start();
   TH.Dispatch(task_fct, NULL);
   TH.WaitForIdle();
   pi = 4.0 * (double)(count[1]+count[2]) / (2*nsamples);
   TR.Stop();
   TR.Report();

   cout << "\n Value of PI = " << pi << endl;
   }
```

LISTING 3.26

CalcPI.C code

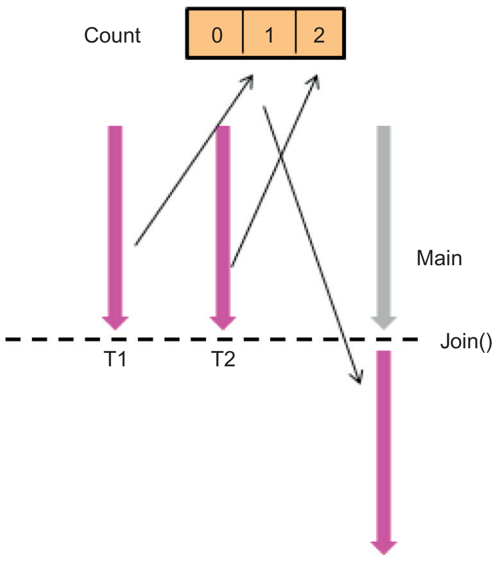


FIGURE 3.4

Worker threads communicating partial results to main().

Here we see for the first time the necessity of communicating partial results obtained by each worker thread, to the master thread. This can be done by introducing a global integer array, count[ ]. The worker thread of rank k writes its partial result at the array element count[k]. Then, the main thread uses these partial results to complete the computation, but *only after returning from the wait that guarantees that all the worker threads have completed their assignment*, to make absolutely sure the count array contains the correct data (a complete explanation of the fact that, after the wait operation, the master thread will read the correct data values from the global array, will be presented in Chapter 7). This looks rather obvious, as illustrated in Figure 3.4.

Notice also that in this computation of π we are just accumulating partial results provided by each thread. This kind of operation, called a *reduction operation*, is very common in applications programming. The partial results can be directly accumulated in a global variable, but, as discussed in Chapter 5, this requires special care. The next chapters will describe general ways of performing reduction operations in multithreaded environments.

The number of threads in CalcPi.C is hardwired to 2. With a reasonably large number of samples, you should be able to see a speedup of 2 for the parallel code, using the O3 compiler option. This speedup is also verified on a dual-core laptop, in spite of the fact that we are using *three threads*: the main thread plus the two worker threads that do the job. This confirms what has often been stated before: while waiting to join the worker threads, the main thread is not using CPU resources. The two cores are therefore fully available for the worker threads.

Example 17: CalcPi.C

To compile, run make calcpi. The number of threads is 2. The number of samples is passed from the command line (the default is 1000000).

3.8.3 THREAD SAFETY ISSUE

When the CalcPi1 code is executed, it yields the correct value of π, with a precision that increases with the number N of samples. More precisely, in a Monte Carlo computation, the precision increases like $\sqrt{N}$: in order to improve the precision by one decimal figure, N must be multiplied by a factor of 100. One can also observe that consecutive executions of the *same code, with the same number of samples* do not return exactly the same value all the time. Results will be slightly different, within the precision of the calculation. This is not very important in this case, because the result is known in advance and can be trusted. In other cases, having an application that provides non-reproducible results may be disturbing. The code we are discussing is not entirely *thread safe*, in the sense that it exhibits some mild dependence on the way threads are scheduled. The next chapter is devoted to a discussion of this issue.

3.9 FIRST LOOK AT OpenMP

A complete discussion of OpenMP is presented in Chapter 10, but it is quite instructive to now take a first look at the way in which the previous SPool based examples can be formulated in OpenMP. The main issue is the replacement of the SPool driven fork-join mechanism by the equivalent fork-join mechanism in OpenMP, based on the parallel directive.

Thread management in OpenMP is implemented with *directives*, followed by a code block to which the directive applies. A directive together with its associated code block is called an OpenMP *construct*.

```
int main(int argc, char **argv)
   {
   ...
   #pragma omp parallel nthreads=2
      {
      // code to be executed by the
      // worker threads
      }
   ...
   }
```

LISTING 3.27

Parallel directive in OpenMP

The listing above shows the parallel construct. When the main thread reaches the parallel construct, it suspends its ongoing task joins the worker team and all together executes the code inside the block. On exit from the block, when all workers have finished, the main thread resumes the suspended execution. Notice that OpenMP does not require a clean cut, well-defined task function inside the code block. The task function will be constructed by the compiler, as long as it is able to understand the programmer intentions. And, in order to make these intentions clear, OpenMP provides a number of *clauses* added to the directive that inform the compiler how a parallel task must be constructed from the code block that follows. The statement nthreads=2 is a clause used to select the number of worker threads (including the master thread). Other clauses have to do with the nature of the data items in the code block (shared? local to a thread?...) All these issues are examined in detail in Chapter 10.

In our examples, we have started thinking parallel from the start, clearly identifying the global variables, and the variables local to the main and the worker threads. In all cases, explicit task functions to be executed by the worker threads are provided. In this context, the migration to OpenMP is straightforward. Here below is the OpenMP version of the CalcPi code, in the source file, CalcPiOmp.C. The code that follows is very close to the SPool code we discussed before, with only three modifications that are explicitly underlined in Listing 3.28.

```
#include <stdlib.h>
#include <TimeReport.h>
#include <iostream>
#include <Rand.h>

// ---- new OPENMP ----
#include <omp.h>
// ---------------

using namespace std;
unsigned long count[2];        // storage of partial results
long nsamples;                 // initialized by command line
Rand R(999);                   // random number generator

void task_fct()                // task executed by each OpenMP thread
   {
   unsigned long ct;
   double x, y;

   // ------- new OPENMP -----
   int rank = omp_get_thread_num();  // get thread rank in [0, nTh-1]
   // -------------------

   ct = 0;
   for(int n=0; n<nsamples; n++)
      {
      x = R->draw();
      y = R->draw();
      if((x*x+y*y) <= 1.0 ) ct++;
      }
```

Continued

```
    count[rank] = ct;           // store partial result un global array
    }

int main(int argc, char **argv)
    {
    TimeReport TR;
    double x, y, pi;

    if(argc==2) nsamples = atoi(argv[1]);
    else nsamples = 1000000;
    TR.StartTiming();

    // -------- new OPENMP -----
    #pragma omp parallel nthreads=2
        {
        task_fct();
        }
    // ---------------------

    pi = 4.0 * (double)(count[0]+count[1]) / (2*nsamples);
    TR.StopTiming();
    TR.ReportTimes();

    cout << "\n Value of PI = " << pi << endl;
    }
```

LISTING 3.28

OpenMP version, CalcPIOmp.C

Here are the modifications introduced in the code:

- The header file omp.h is included, instead of SPool.h.
- In the task function task_fct(), the call to GetRank() is replaced by a call to the OpenMP library function omp_get_thread_num, which returns the thread rank. However, keep in mind that in OpenMP ranks are in the range [0, Nth-1]. The master thread that gets involved in the worker team always carries the rank 0.
- The parallel section is created with the parallel directive. Since all the work of creating the thread function is already done, the parallel code block reduces to a call to task_fct().
- Finally, the size of the count array has been adapted to the fact that now thread ranks are 0 or 1 (instead of 1 and 2 as in the SPool case).

Example 18: CalcPiOmp.C

To compile, run make calcpiomp. The number of threads is 2. The number of samples is passed from the command line (the default is 1000000).

When running this code, a perfect speedup of 2 is observed. There is no difference in performance with respect to the SPool based code.

3.9.1 EXAMPLE: COMPUTING THE AREA UNDER A CURVE

Our next example is another computation of π based on the following result:

$$\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x = \pi$$

This integral is computed by launching N threads. Using a domain decomposition strategy, each worker thread computes the integral in different subdomains of the initial domain $[0, 1]$. As in the previous example, a reduction needs to be performed, to collect the partial results coming from each thread. This problem is handled for the time being in the same way as before: worker threads deposit their results in a global array of doubles, and the main thread completes the computation after the workers team is again idle.

The source code is in the file AreaPi1.C. In order to introduce some new features, we have chosen to leave the number of worker threads open. The main thread gets the number of threads from the command line, with a default value if there is no input from the user. The main differences with the previous example are:

- The size of the global array that will hold the partial results provided by the worker threads is not known. This array is dynamically allocated by the main thread once the number of worker threads is known.
- All threads call a global function—taken from *Numerical Recipes in C*, [16]—that computes the integral of an arbitrary function $f(x)$ in an arbitrary interval $[a, b]$ with given precision. As discussed in the next chapter, this function is thread safe because it has no persistent internal state, and its return values depend only on the argument list.
- Each worker thread determines the limits of its integration domain from the knowledge of the number of threads nTh and its own rank.

Here is a partial listing showing the thread function.

```
// Global data
// ------
double *result;   // this array stores partial results

// Thread function
// ---------
void thread_fct()
   {
```

Continued

```
double size = 1.0/nTh;
int rank    = omp_get_thread_num();
double a    = rank*size;
double b    = (rank+1)*size;
retval = Area(a, b, my_fct, 0.0000001);  // NRC routine
result[rank] = retval;
}
```

LISTING 3.29

Another computation of π

Example 19: AreaPi.C

To compile, run make areapi. The number of threads is read from the command line (the default is 4).

A Spool version of this example is also available. Again, both versions are very close and the differences are easy to understand.

Example 20: AreaPiOmp.C

To compile, run make areapiomp. The number of threads is read from the command line (the default is 4).

3.10 DATABASE SEARCH EXAMPLE

In the programming style adopted up to now, the similarities between the SPool and OpenMP versions of our examples are so evident that one may legitimately wonder about the interest of looking at alternatives to OpenMP. This is indeed the case because the examples examined up to now have a very simple fork-join parallel pattern. The SPool utility will be reconsidered in Chapter 12, including a detailed discussion of what specific parallel contexts may benefit from the innocent looking difference with OpenMP (client threads not getting involved in the parallel job execution). One of the reasons for looking at basic libraries is that OpenMP does not always give programmers direct access to all their possible services, mainly because most of the time they are not needed in applications programming. However, occasionally, some low-level services may help to solve efficiently a given computational problem, as we will have the opportunity of observing in later chapters. One of the reasons for implementing the Spool utility was the necessity of properly canceling a parallel treatment. This is possible in OpenMP only since the latest 4.0 release.

The SPool tools to needed cancel a parallel job are the CancelTeam() and SetCancellationPoint() member functions, used to solve the following problem. Imagine that N worker threads are performing a search in a database. Each thread is performing the same search on independent data sectors. Obviously, once a thread finds the requested data, the parallel search must be stopped. Now, it is possible to formulate this problem by synchronizing the worker threads in such a way that at each step in the search they all know what all the others are doing, and can stop when one of them completes the search. But this approach introduces a substantial synchronization overhead that spoils the parallel performance. It is in fact much more efficient to let all the threads go ahead without any kind of synchronization. The first thread that completes the search calls the CancelTeam() function before terminating. This function smoothly terminates the operation of the worker threads.

The program listed below simulates a database search by launching a number of worker threads that keep retrieving a random number in [0,1] from a random number generator R, until they get a number equal to a given target, called target in Listing 3.30, within a given precision EPS. Each worker thread has its own, local, random number generator, initialized with a unique seed depending on the thread rank. Therefore, each thread retrieves a personalized random number suite, so that there is only one thread that completes the random search first.

The thread function is straightforward: threads enter an infinite loop retrieving random numbers. The purpose of the call to SetCancellationPoint() is to check if a cancellation request has been posted, and if that is the case this function call terminates the ongoing task. The thread that finds the requested target writes the target found as well as its rank in a global structure, and calls CancelTeam(), which terminates the job. The master thread, after the job termination, reads the result of the search from the global structure.

```
struct Data     // data passed to main thread
   {
   double d;
   int    rank;
   };

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

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

    for(;;)
       {
       TS->SetCancellationPoint();
       d = R.draw();
       if(fabs(d-target)<EPS)
          {
          D.d = d;
          D.rank = rank;
          TS->CancelTeam();
```

Continued

```
            }

         }
      }

   int main(int argc, char **argv)
      {
      int nTh;
      if(argc==2) nTh = atoi(argv[1]);
      else nTh = 2;

      TS = new SPool(nTh);          // create workers
      TS->Dispatch(th_fct, NULL);   // launch job
      TS->WaitForIdle();

      std::cout << "\n Received value " << D.d << " from thread "
                << D.rank << std::endl;
      delete TS;
      return 0;
      }
```

LISTING 3.30

DbSearch.C: simulating a database search

The main function is also simple. The number of threads is read from the command line (the default is 2). Increasing the precision (by decreasing EPS) increases the work to be done to find the result. The source file is DbSearch.C.

Example 21: DbSearch.C

To compile, run make dbsearch. The number of threads is read from the command line (the default is 4).

Executing the codes, for the Pthreads version the result comes out in about 1 second. This service uses a native thread cancellation service. Trying to implement this code using standard synchronization tools like barriers, in which each thread checks at each loop iteration what all others are doing, results in unacceptable performances (more than 20 s). The OpenMP 4.0 implementation using the parallel section cancellation service—discussed in Chapter 10—is as efficient as the SPool version proposed here.

3.11 CONCLUSIONS

The simple examples discussed in this chapter have introduced some relevant issues common in multithreaded applications programming. This was first done by using a simple C++ class that encapsulates the basic library protocols for thread creation and termination. However, as shown in the OpenMP versions of the examples, the programming issues are very general and relevant for any programming environment.

We have been able to cope with the communication problems raised in our examples by waiting for worker thread termination before using their partial results. But this is too restrictive. More powerful synchronization tools will be introduced in Chapter 9 that will allow us, among other things, to easily handle data transfers among threads.

3.12 ANNEX: CODING C++11 TIME DURATIONS

The <chrono> header is rather sophisticated, and it proposes a variety of ways of describing *time durations*—intervals of time relative to an initial date, typically the moment at which a function is called—as well as absolute dates in the future. We will meet, as we proceed, a variety of functions that execute *timed waits*, called either wait until(), in which an absolute date for the end of the wait is given, or wait_for() in which the thread waits for a given duration. Absolute dates will never be used in this book, so we restrict ourselves to the description of time durations.

The std::chrono::duration class is defined as follows:

```
template <typename Rep, typename Period>
std::chrono::duration<Rep, Period> const& time_duration;
```

LISTING 3.31

Declaration of the duration class

There are therefore two template parameters that select two types needed to define a duration. The idea is simple: a duration object, called time_duration in the listing above, stores a number of clock ticks:

- The second template parameter is a type that defines the time period between successive ticks, expressed as a fraction of a second. This is achieved by an instance of the std::ration<N, D> class that defines the rational number N/D:
  - std::ratio< 1, 50> is one fiftieth of a second.
  - std::ratio<1, 1000> is one millisecond.
  - std::ratio<60, 1> is one minute.
- The first template parameter is the integer type of the data item that stores the number of ticks (time_duration above): short, int, long, long long.
  - std::chrono::duration<short, std::ratio<1, 50>> count stores in a short the number of one fiftieths of a second.
  - std::chrono::duration<long, std::ratio<60, 1>> stores in a long the number of minutes.

This is the most general way of defining a duration. There are also template specialization that simplify programming, like std::chrono::milliseconds and std::chrono::seconds, that declare integers representing milliseconds and seconds, respectively. They have been used in our previous examples.