CHAPTER

THREAD-SAFE PROGRAMMING

4

4.1 INTRODUCTION

We learned in Chapter 2 that it is perfectly legitimate to have several threads concurrently calling the same library function. A few subtle issues arise, however, in this context. A function is said to be *thread safe* if its behavior, when called by client code in a multithreaded environment, is in conformity with its service specifications. This is, however, not always the case, and this chapter explores different ways of restoring thread safety when this happens.

The concept of thread safety also applies to data sets. It is by no means obvious that a data set—a vector container, for example—will behave according to expectations when concurrently accessed by several threads. The STL containers, in fact, are not thread safe: they do not operate as expected when used directly in a multithreaded environment, and special precautions must be taken. Thread safety for data sets is discussed in-depth in the next chapter.

4.1.1 FIRST EXAMPLE: THE MONTE CARLO π CODE

The `CalcPi` Monte Carlo code yields the correct value of π, with a precision that increases with the number of samples N. However, it was observed that consecutive executions of the *same code, with the same number of samples* do not return *exactly* the same value all the time. Results are slightly different, within the precision of the calculation. This is not very important in this case. But in other cases, having an application returning non-reproducible results may be disturbing. Are we facing a feature, or a bug?

The behavior of the `CalcPi.C` code is easily understood: it arises from the fact that *the two worker threads are accessing the same, global, random number generator*. Indeed, the random number generator `R` is declared with global scope, and its `draw()` function is called by both threads, which is a legitimate action. The problem results from the nature of the random number generator itself.

There is in fact nothing random in a random number generator. The generator returns a *deterministic* suite of numbers *that look random*. The suite is cyclic: after a usually very long period, it repeats itself. Initializing the generator just chooses the starting point of the cyclic suite. In fact, the generator maintains a persistent internal state between successive calls that determines the outcome of the next call. For the particular generator used in the `CalcPi` code, this persistent internal state is carried by the integer variable called `seed`. When the generator is called, the current value of `seed` determines both the returned random number and the new value of `seed` to be used in the next call.

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00004-6

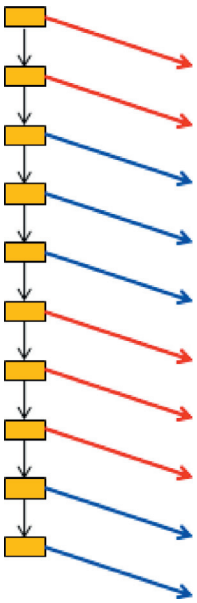

FIGURE 4.1

Two threads calling a global random number generator.

Consider now two threads—a black thread and a gray thread, calling a global generator as shown in Figure 4.1. The generator is represented by the gray box. After returning a value its internal state is changed, and this new internal state determines the next return value. As the figure shows, the generator correctly does its job and delivers to the two threads a unique and deterministic suite of pseudo-random numbers. But the way in which these numbers are distributed among the two threads is not reproducible, because it depends on the particular way in which the threads are scheduled at each run. Since the random numbers received by each thread are not strictly the same at each run, there will be small differences in the acceptance results for each one of them, and this will reflect in small differences in the final result for π.

This is the first example of a *thread safety* issue: the sensitivity of an application to the inner workings of the thread scheduler. Even if in this example the problem can be ignored, this is something that one should definitely be able to control. There are of course clean solutions to this issue, which are the subject of this chapter.

4.2 SOME LIBRARY FUNCTIONS ARE NOT THREAD SAFE

It should be obvious by now that stateless library functions, limited to the manipulation of the argument values passed to the function and local variables sitting in the stack, are thread safe. The stack mechanism for function calls guarantees thread safety in this case. On the other hand, *functions that maintain a persistent internal state between calls*, which records the history of previous calls

FIGURE 4.2

Parsing a string.

to determine the outcome of the next one, *are not thread safe*. They should not be used as such in multithreaded codes, because they cannot be shared among threads. Functions maintaining an internal state can only be used in a *function per thread* basis.

A classical example taken from the C standard library is the strtok function, used to parse a string and to extract tokens from it (which are like the words in a phrase), after defining the characters that separate tokens (like the whitespace in Figure 4.2). The signature of this function is given below.

```
#include <string.h>

char* strtok(char *str,              // the target string
             const char *delim);     // token delimiter chars
```

LISTING 4.1

strtok() function

This function parses a string into a sequence of tokens as follows:

- On the first call to strtok(), the string to be parsed is specified in str. In subsequent calls to parse the same string, the str argument must be NULL.
- The delim argument specifies a set of characters that delimit the tokens in the parsed string.
- Each call to strtok() returns a pointer to a null-terminated string containing the next token. The returned string does not include the delimiting character.
- If no more tokens are found, the function returns a NULL pointer.
- Obviously, strtok() maintains an internal state that records the parsing position in the initial string. *If the function is called again with a non-null* str *pointer, the function forgets about its previous internal state and starts parsing the new string*.

Let us now imagine that thread T1 starts parsing the string str1 and that, before the parsing is finished, thread T2 starts parsing another string str2. When T1 calls strtok() again, it will receive a token from str2, not from str1!. The situation in this case is totally different from the previous Monte Carlo example, where the lack of thread safety was producing slightly different results in the application. Here, the results will be totally wrong.

The C standard library provides *reentrant versions* of many functions that are not thread safe. Reentrant means the functions are thread safe, and can be safely used in a multithreaded environment. The function we have been discussing has a reentrant version called strtok_r(), but with a different signature. Take a look at the man pages (man strtok).

Reentrant library functions are a great help to programmers, but in many cases we have to provide our own reentrant routines. This is the subject we will discuss next.

4.3 DEALING WITH RANDOM NUMBER GENERATORS

Random number generators—widely used in many areas of computational sciences—will be used to discuss in detail the different options for implementing thread safety when dealing with functions with persistent internal state. The objective is the generation of distinct and reproducible sequences of random numbers for each thread participating in the parallel treatment. There are three different ways of attaining this goal, and they will be illustrated with the Monte Carlo computation of π discussed in the previous chapter.

How can a function maintain a persistent state?

In order to understand how functions with persistent internal state operate, let us go back to the very simple generator that produces uniform deviates in [0, 1], used in the Monte Carlo computation of π. We will often use in this book the *Park-Miller minimal standard generator* described in Chapter 7 of [16]. Here is the definition of the generator (for details about the constant values, review the Rand.h include file)

```
double Rand()
   {
   static int seed = 999;
   seed = (seed * IMUL + IADD) & MASK;
   return (seed * SCALE);
   }
```

LISTING 4.2

Simple random number generator

In this function, the persistent internal state is the integer seed. It looks superficially as if it was a local variable, but it is not. Local variables, allocated in the executing thread stack, are destroyed when the function returns. The whole point is the static qualifier of seed , which makes all the difference: the compiler manages in this case to preserve this variable across function calls. Static variables are secretly allocated by the compiler on the heap, and they are accessed through a pointer the compiler stores in some place where it can be recovered every time the function is called. This is what is really happening under the hood, even if the programmer thinks he is manipulating just a special local variable. Another point to be observed is that the initialization statement of the static variable is only executed the first time the function is called. When this generator is accessed by different threads, all the threads share the same seed. This is what makes this function thread unsafe.

Three different strategies can be adopted to restore thread safety for functions with internal persistent state:

- Adopt the standard C library strategy and write a new, stateless reentrant function by changing the signature of the generator in the way discussed in the following.
- Rather than using a global library function accessed by all threads, C++ objects can be used to generate random numbers. It is then possible to allocate a *local* random number generator to each thread. We dispose in this way an independent generator allocated to each parallel task engaged in the computation.
- *Thread local storage* services—proposed by all programming environments—can be used to implement reentrant versions of global, stateful library functions, *without changing their signature*.

This is achieved by modifying the behavior of static variables. They are no longer shared, because these utilities introduce *one static variable per thread*.

The relative merits of all these options will be assessed after discussing the way they operate.

4.3.1 USING STATELESS GENERATORS

In this case, the internal state is taken away from the internal data set of the generator. Instead, it is passed as a new argument in the function call. Each thread allocates its own, proprietary, internal state in a local structure, and passes a pointer to this structure as an argument to the generator, who can then access, use, and modify it. Now, the generator has no memory and no history. It reads and modifies the state received as argument and returns the corresponding random number. If, in addition, each thread performs a personalized initialization of the internal state, each thread will dispose of a personalized and reproducible random number sequence.

A partial listing of the source file McSafe.C is shown below, where the Monte Carlo computation of π is reformulated using a reentrant generator. The listing shows the modified version of the Rand() library function: the signature has changed, and a pointer to the seed to be used is passed as an argument. In the thread function, each thread initializes its local integer seed in a particular way. Different threads have different seeds, multiples of 999 (remember that SPool ranks start at 1). When the code is executed, results are perfectly reproducible.

```
double Rand(int *seed)    // reentrant generator
   {
   *seed = (*seed * IMUL + IADD) & MASK;
   return (*seed * SCALE);
   }

void task_fct(void *P)    // task function
   {
   double x, y;
   long count;

   int seed, rank;
   rank = TH.GetRank();
   seed =999*rank;

   count = 0;
   for(size_t n=0; n<nsamples; n++)
      {
      x = Rand(&seed);
      y = Rand(&seed);
      if((x*x+y*y) <= 1.0 ) count++;
      }
   accepted[rank] = count;
   }
```

LISTING 4.3

Partial listing of McSafe.C—stateless generator

Example 1: McSafe.C

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

4.3.2 USING LOCAL C++ OBJECTS

C++ provides a simple way of handling the problem of disposing of one independent generator per thread. Rather than defining a function with internal state, a class with private data can be defined as follows:

```
class Rand
   {
   private:
    long seed;

   public:
    Rand(long S) : seed(S) {}

    double Draw()
       {
       seed = (seed * IMUL + IADD) & MASK;
       return (seed * SCALE);
       }
   };

...
SPool TH(2);           // global SPool, 2 threads
...

void task_fct(void *P)
   {
   double x, y;
   long count;
   int rank;

   rank = TH.GetRank();
   Rand R(999*(rank+1));
   count = 0;
   for(size_t n=0; n<nsamples; n++)
      {
      x = R.Draw();
      y = R.Draw();
      if((x*x+y*y) <= 1.0 ) count++;
      }
   accepted[rank] = count;
   }
```

LISTING 4.4

Generating random numbers with C++ objects

It is now possible to have in the application as many independent objects instances of the class as we want, each with its own private internal data encapsulating the internal state of the generator. The member function Draw() implements the algorithm that, acting on the private data, returns the random sequence.

If only one global Rand object is used to provide random numbers—as it was done in the previous chapter—then we are in the same trouble as before. But the point now is to create one *local* Rand object per thread, as shown in Listing 4.4. Notice that the internal state is initialized by the constructor of the Rand objects in a thread specific way, using as before the thread rank to define the initial seed of the uniform deviate. This procedure is similar to the previous one, except that now the bookkeeping of passing the internal state to the generator function is implicitly managed by the compiler.

Example 2: McObj.C

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

4.4 THREAD LOCAL STORAGE SERVICES

All multithreaded programming environments provide a *thread local storage* feature—henceforth called TLS—that generalizes static variables to a multithreaded environment. Static variables can now be allocated on a per thread basis, and a function accessed by several threads can dispose of *one independent static variable per thread*. This section presents the way TLS is implemented in the various programming environments.

TLS facilities are not restricted to the enforcement of thread safety in function calls. Chapter 9 will show how TLS utilities, combined with the atomic utilities discussed in Chapter 8, can be used to implement a custom barrier synchronization utility that outperforms the native Pthreads barriers, as demonstrated by the full applications discussed in Chapters 13–15.

TLS utilities operate as follows:

- Thread local variables maintain their value between function calls.
- If the function is called by several threads, the library runtime system *maintains in the heap a copy of one data element per thread* between function calls, with a well-established ownership. A thread always recovers its proprietary data value at all subsequent function calls. This is true no matter how many threads are calling the function.

Before discussing how this feature is implemented in the different programming environments, it is useful to understand how such a mechanism, sketched in Figure 4.3, can operate under the hood. Let us imagine that a given function needs to benefit from thread local storage for a generic data item T (an integer seed, for example, as in our previous examples):

- The application code first requests the creation of a *global container* where T* pointers—integer pointers in our case—will be stored by different threads between successive function calls. This global pointer container is called a key in Pthreads.

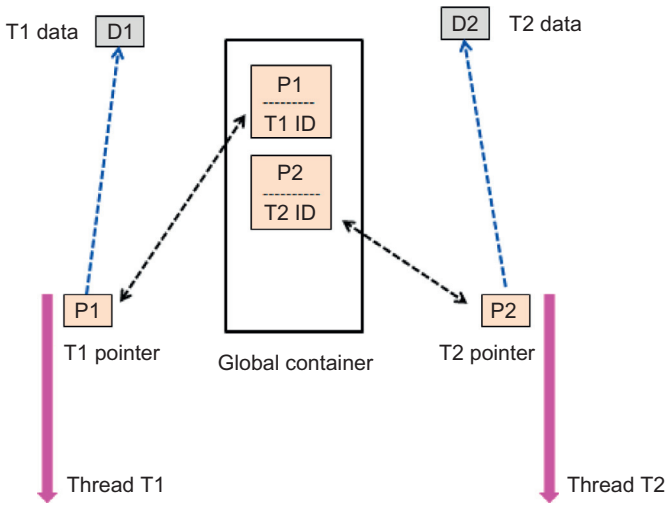

FIGURE 4.3

Thread local storage mechanism.

- There is a first initialization phase where each thread allocates on the heap and initializes its own copy of the persistent data item. In Figure 4.3, thread `T1`—or `T2`—allocates an integer `D1`—or `D2`—at the address `P1`—or `P2`. Once this is done, each thread stores this pointer in the global pointer container.
- At each function call, the thread safe function recovers first the relevant pointer from the container. The container keeps track of pointer ownership, by establishing a map between the stored pointers and the caller thread identity. Each thread executing the function receives in return its proprietary pointer. The function can then read or modify the value of thread the local persistent data item.
- If an application has several different data items requiring TLS, an independent pointer container per data item is required.

The thread local storage interfaces in Pthreads and Windows follow very closely the description given above. In C++11, OpenMP, and TBB the pointer manipulation is hidden to the programmer and the initialization of thread local variables is simpler. We discuss next the specific thread local storage interfaces proposed by the different programming environments.

4.4.1 C++11 THREAD_LOCAL KEYWORD

C++11 has a simple and efficient implementation of thread local storage, based on the `thread_local` keyword, which declares a variable for which an instance per thread is constructed in the application.

In C++11, `thread_local` variables can have different scopes:

- Namespace scope (global scope),
- Static data members of classes,
- local variables inside functions. In this case, they act as a static local variable per thread.

In the first two cases, they are constructed before first use. Local thread_local variables inside a function are initialized the first time as the flow control passes through their declaration in a given thread. Destructors are called when the owner thread returns or calls std::exit().

Thread local variables have, as discussed before, different addresses in different threads. C++11 allows programmers to publish them: a normal pointer to a thread_local variable can be obtained by one thread and passed to another thread. Listing 4.5 shows how the global Rand() generator is rendered thread safe, in the CalcPi.C code:

```
SPool *TH;      // global thread pool

double Rand()
   {
   thread_local int my_seed = 999 * TH->GetRank();
   int retval = (my_seed * IMUL + IADD) & MASK;
   my_seed = retval;
   return (retval * SCALE);
   }

// Task function and main function are the same as in
// CalcPi.C.
```

LISTING 4.5

McSafe_S.C (partial listing)

Look at the declaration and initialization of the thread_local my_seed in the Rand() function. There is a substantial amount of intelligence in this unique statement: it is executed *the first time that a new thread calls the function*. Moreover, a thread specific initialization of my_seed, proportional to the rank of the caller thread, is implemented. In the Monte Carlo computation that follows, each thread receives a personalized, reproducible suite of uniform deviates.

Example 3: McSafe_S.C

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

4.4.2 OpenMP THREADPRIVATE DIRECTIVE

OpenMP also introduces another elegant programming interface for TLS: a threadprivate directive that promotes a static local variable to a TLS status. Again, here are the very simple modifications that need to be introduced in the unsafe random number generator Rand() to make the CalcPiOmp.C example (Listing 3.28 in Chapter 3) thread safe. The source code is McSafeOmp.C:

```
double Rand()
   {
   static int seed = 999 * omp_get_thread_num();
   #pragma omp threadprivate(seed)
```

Continued

```
    seed = (seed * IMUL + IADD) & MASK;
    return (seed * SCALE);
    }
```

LISTING 4.6

Partial listing of McSafeOmp.C

All that has happened here is that a directive has been added to the unsafe version of the Rand() function, which promotes the previously declared static variables seed to a threadprivate status.

Example 4: McSafeOmp.C

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

4.4.3 WINDOWS TLS SERVICES

Windows has a simple TLS service, with programming interfaces that closely follow the description shown in Figure 4.3. For each local variable that requires TLS, a container of pointers is requested to the operating system, which returns an index identifying the container. Then, each thread uses this index to store or retrieve the pointer referring to the thread local instance of the variable, *cast as void pointers*. Here is the Windows TLS programming interface, as described in the Windows API online documentation [13]:

- DWORD TlsAlloc(). This function call returns an index that identifies the global pointer container, seen by all threads. Any thread in the process can subsequently use this index to store and retrieve its proprietary pointer.
- BOOL TlsSetValue(DWORD index, LPVOID ptr): Stores the pointer value ptr in the container referenced by index. Threads make this call to store their local pointer value when the local variable is first allocated. Returns false in case of failure.
- LPVOID TlsGetValue(DWORD index): Threads call this function to retrieve their pointer value at each function call.
- BOOL TlsFree(DWORD index): Releases the pointer container when the service is no longer needed.

Listing 4.7 shows how the above programming interface is used in the computation pf π. The main difference with the C++11 or OpenMP environments is that there is no automatic initialization of thread local variables the first time the generator is accessed. Allocation, initialization, and storage of proprietary pointers must be done explicitly. In the program below, tlsIndex is the index returned by the kernel, identifying the pointer container. This index is initialized by main() when the program starts.

The InitPointer() function listed below is an initialization task performed by all threads, which allocate and initialize their individual seeds, and store the corresponding pointers in the global pointer container. The only modification to the Rand() function is the first statement in the function body, in which the proprietary pointer is retrieved from the pointer container. The task function computing π is not modified. Finally, notice that the main() function *launches two parallel jobs*: the pointer initialization job and the actual computation.

```
// Global variables
// – – – – – – – –
SPool *TH;
...
DWORD tlsIndex;    // Windows TLS container
...
void InitPointer(void *P)   // Initialization parallel job
   {
   int *seedptr;
   int rank = TH->GetRank();

   // Allocate seed in the heap, and store pointer in
   // tlsIndex
   // – – – – – – – – – – – – – – – – – – – – – – –
   seedptr = new int(999*rank);
   TlsSetValue( tlsIndex, static_cast<void*>(seedptr));
   }

double Rand()  // modified generator
   {
   int *my_seed = static_cast<int*>(TlsGetValue(tlsIndex));
   int retval = (*my_seed * IMUL + IADD) & MASK;
   *my_seed = retval;
   return (retval * SCALE);
   }

// The main function
// – – – – – – – – –

int main(int argc, char **argv)
   {
   long C;
   ...
   tlsIndex = TlsAlloc();   // initialize global container
   ...
   // Initialize thread specific pointers
   // – – – – – – – – – – – – – – – – – –
   TH->Dispatch(InitPointer, NULL);
   TH->WaitForIdle();

   // Perform the computation
   // – – – – – – – – – – – –
   TH->Dispatch(task_fct, NULL);
   TH->WaitForIdle();

   C = 0;
   for(int n=1; n<=nTh; ++n) C += accepted[n];
   double pi = 4.0 * (double)C / (nsamples*nTh);
   cout << "\n Value of PI = " << pi << endl;
   }
```

LISTING 4.7

Partial listing of McSafe_W.C

Example 5: McSafe_W.C

To compile, run nmake mcsafe_w. The number of threads is 2. The number of samples is read from the command line (the default is 1000000).

4.4.4 THREAD LOCAL CLASSES IN TBB

TBB implements TLS by following the same strategy described before, but the programming interfaces are slightly different: an explicit initialization of thread local variables is not needed, and there are no explicit pointer manipulations. The TBB documentation on this subject can be found in the "Thread Local Storage" topic [17].

TBB has two different implementations of TLS in two class templates, the template parameter T being the type of the thread local data variable to be stored. Object instances of these classes are the global containers that store the different T* pointers. Both TBB classes implement TLS, but they differ in the nature of some additional services they provide. The two classes are:

- combinable<T>. This class is useful when a reduction of the different instances of the thread local variable needs to be performed.
- enumerable_thread_specific<T>. This class enables access to the global container holding the different instances of the thread local variable, as an STL container, with iterators, in order to perform more sophisticated operations on this data set.

Rather than discussing the classes in detail, the simple example discussed in the previous sections is reexamined, and one of the TBB classes is used as a standalone utility, to make the Rand() generator thread safe. The rest of the code is the same as in CalcPi.C. With this example in mind, it is easy to understand the TBB documentation to get more information if needed [18].

The code source is McSafeTbb.C, partially listed below. Allocations and initializations of the thread local variables are done implicitly by the library—as it was the case in C++11 and OpenMP. All that is required here is the definition of a function taking no arguments and returning a T object—an integer in our case—that is used by the library to initialize each new instance of the thread local variable. This function is called finit() in Listing 4.8. The library knows about it because a pointer to this function is passed via the constructor of the global pointer container.

```
#include <SPool.h>
#include <tbb/enumerable_thread_specific.h>
...
int finit();            // forward declaration
...
SPool  TH(2);           // global variables
long   nSamples;
tbb::enumerable_thread_specific<int> seed(finit);

int finit()             // auxiliary initialization function
```

```
    {
    int retval;
    retval = 999 * TH.GetRank();
    return retval;
    }

// The random number generator
// - - - - - - - - - - - - - --
double Rand()
    {
    tbb::enumerable_thread_specific<int>::reference my_seed = seed.local();
    int retval = (my_seed * IMUL + IADD) & MASK;
    my_seed = retval;
    return (retval * SCALE);
    }
```

LISTING 4.8

Partial listing of McSafeTbb.C

There are a number of subtleties in this code that need careful discussion. First, the header file enumerable_thread_specific.h needs to be included. Notice that:

- A seed object of type enumerable_thread_specific <int> needs to be declared with global scope. This global seed object is in fact the container that stores the different instances of the thread local integer variable.
- The object constructor receives as argument the pointer to a function—called finit() in the code above—that TBB will use to initialize the different instances of the seed for each thread, if and when they will be created. If no argument is passed to this constructor, a default initialization is used.
- At this point, the seed container is empty. The main() function has not yet started, and the worker threads are not yet created.
- The real action is in the first code line in the body of the Rand() function, which declares the my_seed object. This object is a reference to the local instance of the seed, returned by the local() member function of the class.
- The TBB documentation specifies that the thread local instance of seed stored in the container *is created and initialized the first time the* local() *member function is called by each thread*. This is similar to the way C++11 and OpenMP work.
- At this point, we dispose of my_seed, which is a *reference*—i.e., an alias—of the thread local variable stored in the container. Changes in my_seed automatically change the values of the stored thread local variable.

The initialization function finit() performs the same initialization used before in other versions of this example: the initial generator seed is 999 times the rank of the caller thread, which, in this case, is always bigger or equal to 1.

Example 6: McSafeTbb.C

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

Executing this code returns exactly the same results for the variance as the previous thread safe versions of this example. Notice that this example is portable, since it only depends on TBB and the vath library. It can then be run as such on any Linux or Windows environment.

4.4.5 THREAD LOCAL STORAGE IN PTHREADS

The Pthreads library proposes the keys service as a very general way of generating and managing thread specific data items. This service closely parallels the general description above, with the key object playing the role of the global pointer container.

Some people consider this service clumsy and inefficient. The point is that, written in C, Pthreads utilities are often rather verbose, and programmers must perform explicitly some actions—initializations, or explicit pointer manipulations—that are implicit in all the other programming environments we have just discussed.

If you absolutely need to understand the Pthreads keys service, take look at the McSafe_P.C source code, and look at the way the Rand() function is rendered thread safe. For more details on the keys service, look at the Pthreads documentation. Another interesting alternative for Pthreads programmers is to use the standalone TBB thread local class in a Pthreads environment.

Example 7: McSafe_P.C

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

4.5 SECOND EXAMPLE: A GAUSSIAN RANDOM GENERATOR

The internal persistent state engaged in the previous example is very simple: just an integer variable. In order to make it clear how to manage a more complex situation, another example is included, in which the internal persistent state is composed of three variables of different types.

This application is simple: testing the behavior of a generator of Gaussian deviates with variance 1. A Gaussian generator returns a double on the real axis. The probability of getting a return value in the tiny interval $(x, d + dx)$ is

$$P(x)\,\mathrm{d}x = \frac{1}{\sqrt{(\pi)}} \exp -x^2\,\mathrm{d}x$$

Obviously, the probability of getting *any* value on the real axis is 1:

$$\int_{-\infty}^{\infty} P(x)\,\mathrm{d}x = 1$$

Let us assume that we keep getting Gaussian deviates from the generator and marking their values on the real axis. The density of points so obtained follows the Gaussian law indicated above: it is mainly concentrated around the origin, with a characteristic width. A rough idea of the width of the Gaussian distribution is provided by the mean value of x^2. This is done by getting N samples of x from the generator and computing the mean value mv as

$$mv = \frac{1}{N}\sum_{i=1}^{N} x_i^2$$

The computation is done using two threads, each one computing the above mean value for N/2 samples. The main() function recovers the partial results from each thread, and averages them. This is the final result, which should be equal to one.

Box-Muller algorithm

The Box-Muller algorithm for generating Gaussian deviates consists of a mapping that transform two uniform deviates in $[0, 1]$ into two Gaussian deviates. Gaussian deviates are therefore generated by pairs. A function GRand() producing Gaussian deviates is shown in Listing 4.9.

```
double Grand()
   {
   static double ransave = 0.0;   // internal state variable
   static int flag = 0;           // internal state variable
   double x1, x2, scratch;

   if(flag)
      {
      flag = 0;
      return ransave;
      }
   else
      {
      x1 = Rand();
      x2 = Rand();
      scratch = sqrt(-2*log(x1));
      x1 = cos(2 * PI * x2);
      x2 = sin(2 * PI * x2);
      ransave = scratch * x2;
      flag = 1;
      return(scratch * x1);
      }
   }
```

LISTING 4.9

Box-Muller algorithm

This Gaussian generator maintains a persistent internal state for two reasons:

- It uses internally the very simple uniform generator in [0, 1] used before, and it needs an integer seed for it. This seed is the persistent internal state that controls the operation of the uniform generator.
- Gaussian deviates are therefore generated by pairs. One value is returned to the caller and the other is reserved for the next call. Besides the seed, the internal state is therefore defined by:
  - a double randsave, which stores the possible return value for the next call,
  - an integer flag, which indicates if a precomputed return value is available in ransave.
- If a return value is available, the generator returns it. Otherwise, two values are produced; one is returned and the other is stored for the next call.

Gaussian examples

Here is a list of the different versions of the Gaussian generator examples. In all cases, the number of threads is hardwired to 2, and the number of samples used in the computation can be passed by the command line (the default value is 10000000).

- GrUnsafe.C: the straightforward, thread unsafe version of the example. To compile, run *make grunsafe*.
- GrSafe.C: the reentrant, stateless version of the generator. To compile, run *make grsafe*.
- GrObj.C: the version based on a local C++ class. To compile, run *make grobj*.
- GrWin.C: the version based on the Windows TLS service. To compile, run *nmake grwin*.
- GrOmp.C: the version using OpenMP the threadprivate directive. To compile, run *make gromp*.
- GrTbb.C: the version based on the TBB thread local storage facility. To compile, run *make grtbb*.

4.6 COMMENTS ON THREAD LOCAL STORAGE

Thread local storage provides an elegant and efficient way of implementing thread safety for functions with internal state. This looks like the perfect solution to the initial problem of providing one stateful function per thread. This is a robust solution *as long as one has a well-identified, distinct parallel task per thread*.

This has indeed been the case in all the examples discussed so far. OpenMP parallel sections and our SPool utility establish a one-to-one mapping between the parallel tasks that are using the random number generators, and the worker threads that execute them. But this does not need to be the case in general. Starting from Chapter 10, more general parallel constructs will be discussed in which it is possible to have more independent tasks than active threads. In this case, the task to thread mapping is not one to one. Threads take over new tasks when they are ready, according to well-established scheduling strategies. Threads can also suspend ongoing tasks to serve other tasks waiting for execution. In TBB and OpenMP, the task to thread mapping has a high level of sophistication. It is an *unfair* mapping, in the sense that it does not necessarily operate on a first come, first served basis.

If the task to thread mapping is not one to one, the thread local storage mechanism does not guarantee thread safety. *Thread safety requires one independent stateful function per task, not per*

thread. If more than one task is executed by the same thread, thread safety is broken because several stateful functions are sharing the same thread local internal state in a non-reproducible way.

The only robust strategy that works in all cases is implementing one stateful function *per task*, which is exactly what was done in the first two options. In these cases, persistent internal state is owned by a task, not by a thread. The most efficient and universal solution to the problem discussed in this chapter is then using task local C++ objects with internal state, or its poor man version provided by the first option discussed above.

4.7 CONCLUSION

This chapter has proposed a thorough discussion of thread safety in the context of function calls. This is not the end of it, there are very critical thread safety issues related to the access of global data structures in a multithreaded environment. This is the subject of the next chapters.