CHAPTER 10

Threading

The subject of multithreading is a pretty big one, and worthy of several books all to itself [Bute1997, Rich1997]. In an attempt to simplify, I intend to stipulate that the challenges of multithreaded programming all pertain to synchronization of access to resources. The resources may be a single variable, a class, or they may even be some product of one thread that is to be consumed by another. The point is that if two or more threads need to access the same resource their access needs to be made safe. Alas, both C and C++ were developed before multithreading became as popular as it now is. Hence:

Imperfection: C and C++ say nothing about threading.

What does that mean? Well, in no part of the standard will you find any reference to threading.[1] Does this mean that you cannot write multithreaded programs with C++? Certainly not. But it does mean that C++ provides no support for multithreaded programming. The practical consequences of this are considerable.

The two classic concepts [Bute1997, Rich1997] one must be wary of when writing multitasking systems are race-conditions and deadlocks. Race-conditions occur when two separate threads of execution have access to the same resource at the same time. Note that I'm using the term "thread of execution" here to include processes on the same system and threads in the same process or in different processes in the same host system.

To protect against race-conditions, multitasking systems use synchronization mechanisms, such as mutexes, condition variables, and semaphores [Bute1997, Rich1997], to prevent concurrent access to shared resources. When one thread of execution has acquired a resource (also called locking the resource), other threads are locked out, and go into a wait state until the resource is released (unlocked).

Naturally the interaction of two or more independent threads of execution is potentially highly complex, and it is possible to have two threads each holding a resource and waiting for the other to release. This is known as a deadlock. Less common, but just as deadly, is *livelock*, whereby two or more processes are constantly changing state in response to changes in the others and so no progress can be made.

Both race conditions and deadlocks are hard to predict, or test for, which is one of the practical challenges of multithreaded programming. Even though deadlocks are very easy to detect—your executable stops—they are still hard to diagnose, since your process (or a thread within it) has hung.

---

[1]The only occurrence of the word *thread* is (C++-98: 15.1;2) when discussing threads of control in exception-handling.

10.1 Synchronizing Integer Access

Since the contents of the processor registers are stored in the thread context each time a process experiences a thread switch, the most basic form of synchronization is that which ensures that access to a single memory location is serialized. If the size of the memory to read at that location is a single byte, then the processor will access it atomically. Indeed, the same would apply to the reading of larger values according to the rules of the given architecture. For example, a 32-bit processor would ensure that accessing a 32-bit value was serialized, so long as the value was aligned on a 32-bit boundary. Serialization of access to nonaligned data may or may not be provided by any given processor. It's hard to imagine a workable architecture where such atomic operations were not provided.

The processor's guarantees are fine if you want to read or write platform-sized integers atomically, but there are many operations that you would wish to have atomic that are not so because they are actually several operations in one. The classic one of these is incrementing or decrementing variables. The statement

```
++i;
```

is really only a shorthand for

```
i = i + 1;
```

The increment of `i` involves a fetch of its value from memory, adding `1` to that value, and then storing the new value back in `i`'s memory location, known as *Read-Modify-Write* (RMW) [Gerb2002]. Since this is a three-step process, any other thread that is concurrently attempting to manipulate the value of `i` can cause the result in either or both thread(s) to be invalidated. If both threads are attempting to increment `i`, it is possible for the threads to be out of step, as in:

```
Thread 1                  Thread 2
load i from memory
                          load i from memory
                          increment value
increment value
                          store new value to i
store new value to i
```

Both threads loaded the same value from memory, and when Thread 1 stores the incremented value, it writes over the result of Thread 2. The net effect is that one of the increments is lost.

In practice, different processors will have different operations for conducting these steps. For example, on the Intel processor it could be implemented as follows:

```
mov eax, dword ptr [i]
inc eax
mov dword ptr [i], eax
```

or, more succinctly, as

```
add dword ptr [i],1
```

Even in the second form, where there is a single instruction, the operation is not guaranteed atomic on multiprocessor systems, because it is logically equivalent to the first, and a thread on another processor may interleave in the manner described above.

10.1.1 Operating System Functions

Since atomic increments and decrements are the cornerstone of many important mechanisms, including reference counting, it is very important that there are facilities for conducting these operations in a thread-safe manner. As we will see later, atomic integer operations can often be all that is required to make nontrivial components thread safe,[2] which can afford considerable performance savings.

Win32 provides the `InterlockedIncrement()` and `InterlockedDecrement()` system functions, which look like the following

```
LONG InterlockedIncrement(LONG *p);
LONG InterlockedDecrement(LONG *p);
```

These implement preincrement and predecrement semantics. In other words, the return value reflects the new value, rather than the old. Linux provides analogous functions [Rubi2001]: `atomic_inc_and_test()` and `atomic_dec_and_test()`. Similar functions may be available on a platform-specific basis.

Using such functions we can now rewrite our initial increment statement in a thoroughly thread-safe manner

```
atomic_inc_and_test(&i); // ++i
```

The implementation of such a function for the Intel processor would simply incorporate the `LOCK` instruction prefix, as in:[3]

```
lock add dword ptr [i], 1
```

The `LOCK` instruction prefix causes a `LOCK#` signal to be expressed on the bus, and prevents any other threads from affecting that memory location for the duration of the `ADD` instruction. (Naturally, it's a lot more complex than this, involving cache lines and all kinds of magic jiggery-pokery, but logically it makes the instruction atomic with respect to any other threads/processors.[4])

---

[2]The Synesis BufferStore component that I mentioned in Chapter 7 is one example, deriving its high speed from the avoidance of any kernel object synchronization.

[3]This is not the actual instruction(s), but I'm trying to be brief!

[4]On multi-CPU machines, or machines with hyperthreading/multiple-core CPUs, threads may really be running in parallel, whereas on single CPUs threads only appear to execute simultaneously. In single-CPU machines, interrupts may interrupt instructions (except for atomic ones).

The downside to applying locking semantics is a cost in speed. The actual costs will differ for different architectures: on Win32 the cost can be roughly 200–500% that of the nonlocked variant (as we'll see later in this chapter). Thus, it is not desirable simply to make every operation thread safe; indeed, the whole point of multithreading is to allow independent processing to take place concurrently.

In practice, therefore, one normally determines whether to use atomic operations dependent on a build setting. In UNIX, one tends to use the `_REENTRANT` preprocessor symbol definition to indicate to C and C++ code that the link unit is to be built for multithreading. In Win32 it is `_MT` or `_ _MT_ _` or similar, depending on the compiler. Naturally, such things are abstracted into a platform/compiler-independent symbol, for example, `ACMELIB_MULTI_THREADED`, which is then used to select the appropriate operations at compile time.

```
#ifdef ACMELIB_MULTI_THREADED
  atomic_increment(&i);
#else /* ? ACMELIB_MULTI_THREADED */
  ++i;
#endif /* ACMELIB_MULTI_THREADED */
```

Since this is ugly stuff to be dotted around, it's also common practice to abstract the operation into a common function within which the preprocessor discrimination can be encapsulated. Plenty of examples of this exist in common libraries, such as Boost and Microsoft's Active Template Library (ATL).

I should point out that not all operating systems provide atomic integer operations, in which case you may need to resort to using an operating system synchronization object, for example, a mutex, to lock the access to the atomic integer API, as we'll look at later in the chapter.

10.1.2 Atomic Types

We've seen how we can simply use – – and ++ on the integer types for single threaded contexts, but for multithreading we need to use operating system/architecture primitives. The downside of this approach is that even when we abstract the differences into a common function, say `integer_increment`, it relies on all uses of the integer being done atomically. It's not terribly difficult to forget one, in which case you could have a race condition in your application that's very difficult to diagnose.

C++ is a language that aims to provide uniformity of syntax by allowing user-defined types to appear as built-in types. So why not make atomic types that look like built-in types, except that they operate atomically, in all respects? There's no reason why not, and it's actually very simple to do:[5]

Listing 10.1

```
class atomic_integer
{
```

[5]The `volatile` qualifiers in the class definition facilitate the use of `volatile` as a qualifier on the declaration of any such variables, which is a reasonably common practice in multithreaded code, since it prevents the compiler from manipulating variables in its internal registers, thus failing to synchronize the value with the actual memory location of the variables. We'll look at it in more detail in section 18.5.

```
public:
  atomic_integer(int value)
    : m_value(value)
  {}
// Operations
public:
  atomic_integer volatile &operator ++() volatile
  {
    atomic_increment(&m_value);
    return *this;
  }
  const atomic_integer volatile operator ++(int) volatile
  {
    return atomic_integer(atomic_postincrement(&m_value));
  }
  atomic_integer volatile &operator —() volatile;
  const atomic_integer volatile operator —(int) volatile;

  atomic_integer volatile &operator +=(value_type const &value) volatile
  {
    atomic_postadd(&m_value, value);
    return *this;
  }
  atomic_integer volatile &operator -=(value_type const &value) volatile;

private:
  volatile int m_value;
};
```

This is a clear area in which C++ makes threading easier and simpler. However, there is a question as to how much of the natural semantics of integer types are made available. The code above shows how easy it is, given a library of atomic integer functions, to implement increment/decrement, and addition and subtraction. However, doing multiplication, division, logical operations, shifting, and other operations are considerably more complex, and most atomic integer libraries do not provide such operations. If you want them in your code, that'll have to be the author's get-out-of-jail-free card: an exercise for the reader.

The Boost atomic operations components take just this approach, providing platform-specific versions of a type, `atomic_count`, which provides only ++ and – operations (`atomic_increment`/`atomic_decrement`), along with implicit conversion (`atomic_read`), so if you choose to opt out of anything much more complicated, you'll be in good company.

10.2 Synchronizing Block Access: Critical Regions

For most synchronization requirements, a single atomic operation is not sufficient. Such cases require exclusive access to what is known as a critical region [Bulk1999]. For example, if you have two variables to update atomically, you must use a synchronization object to ensure each thread is granted exclusive access to the critical region, as in:

```
// Shared objects
SYNC_TYPE   sync_obj;
SomeClass   shared_instance;
. . .
// Somewhere in code called by multiple threads
lock(sync_obj);
int i = shared_instance.Method1(. . .);
shared_instance.Method2(i + 1, . . .);
unlock(sync_obj);
```

The two operations `Method1()` and `Method2()` must be conducted in an uninterrupted sequence. Hence, the code wherein they are called is encapsulated within calls to acquire and release a synchronization object. In general, the use of all such synchronization objects is very expensive, and so ways in which their costs can be minimized or avoided are desirable.

There are two causes of the high-costs of synchronization objects to secure critical regions. The first is that the costs of using the synchronization objects themselves can be high. For example, consider the timings (in milliseconds) shown in Table 10.1, for 10 million acquire-release cycles of four Win32 synchronization objects, and a control scenario consisting of two empty function calls. The results plainly show that the cost of using synchronization objects is considerable, up to 150 times that of a normal function call.

The second cost associated with synchronization objects for protecting critical regions is that incurred by any threads that are blocked from entering the critical region. The longer the critical region, the more likely this cost is to be incurred, and therefore it is good to keep critical regions as short as possible or to break them into subcritical sections, as was discussed in section 6.2. However, because the costs of the acquire and/or release calls can be very high, the balance between breaking critical regions and the causing long waits for pending threads is a delicate one. Only performance profiling can give you definitive answers on a case-by-case basis.

10.2.1 Interprocess and Intraprocess Mutexes

Mutexes are the most common form of synchronization object for guarding critical regions and, depending on the operating system, there can be two kinds: interprocess and intraprocess. An interprocess mutex is one that may be referenced in more than one process, and can therefore provide interprocess synchronization. On Win32, such a mutex is normally created by

Table 10.1

Synchronization Object	Uniprocessor machine	SMP machine
None (control)	117	172
CRITICAL_SECTION	1740	831
Atomic inc	1722	914
Mutex	17891	22187
Semaphore	18235	22271
Event	17847	22130

calling `CreateMutex()` and naming the object. Other processes may then access the same mutex by specifying the same name to `CreateMutex()` or `OpenMutex()`.[6] With PTHREADS [Bute1997], the UNIX POSIX standard threading library, a mutex may be passed to its child via a process fork, or can be shared via mapped memory.

Intraprocess mutexes, by contrast, are only visible to threads within a process. As they do not need to exist across process borders, they can partially or completely avoid the costly trips to the kernel, since their state can be maintained within process memory. Win32 has such a construct, known as a `CRITICAL_SECTION`, which is a lightweight mechanism that keeps most of the processing out of the kernel, only making kernel calls when the ownership is to be transferred to another thread. There are considerable performance gains to be had by using intraprocess mutexes where suitable, as can be seen in Table 10.1, whose results were obtained by an executable with a single thread. We'll see later how the `CRITICAL_SECTION` performs when the application is multithreaded.

10.2.2 Spin Mutexes

There's a special kind of intraprocess mutex, which is based on the ordinarily bad practice of polling. Simply, polling is waiting for a condition to change by repeatedly testing it, as in:

```
int g_flag;

// Waiting thread
while(0 == g_flag)
{}
. . . // Now do what we've been waiting for
```

This kind of thing chews up the cycles, as the polling thread is often[7] given equal priority with the thread that will be changing the flag to enable it to proceed. Polling is one of those *bad ideas* that ordinarily mark one out as a multithreading neophyte, suitable for apple-pie desks or unlooked-for severance pay.

However, there are circumstances in which spinning is eminently suitable. Let's first look at an implementation of a spin mutex, the imaginatively named UNIXSTL[8] class `spin_mutex`, shown in Listing 10.2.

Listing 10.2

```
class spin_mutex
{
public:
  explicit spin_mutex(sint32_t *p = NULL)
    : m_spinCount((NULL != p) ? p : &m_internalCount)
    , m_internalCount(0)
  {}
```

[6] An unnamed mutex handle may also be passed to a child process through other IPC mechanisms, but naming is the most straightforward mechanism.

[7] Depending on respective thread priorities and on any external events on which other threads may be waiting.

[8] The STLSoft subproject that maps UNIX APIs to STL.

```
  void lock()
  {
    for(; 0 != atomic_write(m_spinCount, 1); sched_yield())
    {}
  }
  void unlock()
  {
    atomic_set(m_spinCount, 0);
  }
// Members
private:
  sint32_t  *m_spinCount;
  sint32_t  m_internalCount;
// Not to be implemented
private:
    spin_mutex(spin_mutex const &rhs);
    spin_mutex &operator =(spin_mutex const &rhs);
};
```

The mechanism of a spin mutex is very simple. When `lock()` is called, an atomic write is performed to set the spin variable `*m_spinCount` (an integer) to 1. If its previous value was 0, then the caller was the first thread to set it, and has "acquired" the mutex, so the method then returns. If its previous value was 1, then the caller was beaten to it, by another thread, and thus does not own the mutex. It then calls the PTHREADS function `sched_yield()` to yield to another thread, and when it wakes up again, it tries again. It repeats this until it succeeds. Thus it is locked out of ownership of the mutex.

When the thread that acquired the mutex calls `unlock()`, the spin variable is set back to 0, and another thread may then acquire it. The slight complication with the constructor and the `m_internalCount` is that this class can be constructed on an external spin variable, which can be very useful in certain circumstances (as we'll see in Chapters 11 and 31).

Spin mutexes are not a good solution where there is a high degree of contention, but where the likelihood of contention is very low and/or the cost of acquiring/releasing the synchronization mechanism must be low, they can be an effective solution. Given their potential high cost, I tend to use them only for initialization where contention is exceedingly rare, but theoretically possible, and must be accounted for. Also, they are not "reentrant," that is they cannot be acquired multiple times from the same thread. Attempting to do so results in deadlock.

10.3 Atomic Integer Performance

Before we and move on to multithreading extensions (section 10.4) and *Thread Specific Storage* (section 10.5), I want to take a look at the performance aspects of various atomic integer operation strategies.

10.3.1 Atomic Integer by Mutex

Where your atomic integer operations are not provided by your operating system, you may need to resort to using a mutex to lock the access to the atomic integer API, as shown in Listing 10.3.

Listing 10.3

```
namespace
{
  Mutex s_mx;
}
int atomic_postincrement(int volatile *p)
{
  lock_scope<Mutex>   lock(s_mx);
  return *p++;
}
int atomic_predecrement(int volatile *p)
{
  lock_scope<Mutex>   lock(s_mx);
  return --*p;
}
```

The problem here is performance. Not only do you pay the sometimes considerable cost of going to the kernel in the acquiring and release of the mutex object, but you also have contention from several threads wanting to perform their atomic operations simultaneously. Every single atomic operation within your process involves the single mutex object, which naturally leads to a bottleneck.

I once witnessed a tragic attempt to ameliorate this cost by having a separate mutex for each atomic function. Unfortunately, this proved very successful in reducing waiting times. As I'm sure you've guessed, this was thoroughly tested on a single-processor Intel machine. As soon as the application was run on a multiprocessor machine, it fell in a heap.[9] Since each mutex protected the function, rather than the data, it was possible to have some threads incrementing a variable while another was decrementing it. All that was achieved was to prevent two threads doing the same thing to the same integer at the same time. As with so many things in multithreading, you cannot have confidence in your code until you've tested it on a multiprocessor machine.

Despite that abject failure, there is a way to share the contention between more than one mutex in order to reduce the bottleneck. What is required is to base the mutex selection on a property of the variable being manipulated. Well, there's only one attribute we know about it: its address. (We can't very well know its value, since that is going to be changing.)

This looks something like the following:

```
namespace
{
  Mutex        s_mxs[NUM_MUTEXES];
};

int __stdcall Atomic_PreIncrement_By(int volatile *v)
{
  size_t             index = index_from_ptr(v, NUM_MUTEXES);
```

[9]It would suffer the same fate on modern hyperthreading single processor machines.

```
  lock_scope<Mutex> lock(s_mxs[index]);
  return ++*(v);
}
```

The function `index_from_ptr()` provides a deterministic mapping from an address to an integer in the range `[0, NUM_MUTEXES-1)`. It would not be suitable simply to perform modulo division of the address, since most systems align data on boundaries of 4, 8, 16, or 32 bytes. Something like the following might be suitable:

```
inline size_t index_from_ptr(void volatile *v, size_t range)
{
  return (((unsigned)v) >> 7) % range;
}
```

In testing on my own Win32 machine, I found 7 to give better performance than other values, but that's unlikely to translate to other platforms, so you'd want to optimize this for each platform.

10.3.2 Run Time Architecture Dispatching

I'd like to show you a little trick for enhancing performance of atomic integer operations for the Intel platform. As we've learned, the Intel processor will conduct a single-instruction RMW operation (such as ADD, XADD) uninterrupted, so on single processor machines a bus lock is not required. Conversely, the bus must be locked for these operations on multiprocessor machines. Since it's much easier to build and ship a single version of code, it would be nice for our code to only pay the performance cost of bus locking when necessary. Since the number of instructions in either case is very small, we'd need a very efficient way of doing this, otherwise the test would cause more latency than the savings gained. The solution looks like the simplified form[10] shown in Listing 10.4, which is compatible with most modern Win32 compilers.

Listing 10.4

```
namespace
{
  static bool s_uniprocessor = is_host_up();
}

inline __declspec(naked) void __stdcall
  atomic_increment(sint32_t volatile * /* pl */)
{
  if(s_uniprocessor)
  {
    _asm
    {
      mov ecx, dword ptr [esp + 4]
      add dword ptr [ecx], 1
```

[10]The full implementations for these functions are to be found in the STLSoft libraries, which are included on the CD.

```
      ret 4
    }
  }
  else
  {
    _asm
    {
      mov ecx, dword ptr [esp + 4]
      lock add dword ptr [ecx], 1
      ret 4
    }
  }
}
```

Even if you're not familiar with the Intel assembler, you should be able to understand how simple the mechanism is. The `s_uniprocessor` is true for uniprocessor machines, and false for multiprocessor machines. If it's true, the increment is effected without a lock. If it's false, the lock is used. Any possible race conditions in the instantiation of `s_uniprocessor` are unimportant, since the default case is to apply the lock, which is benign.

In the performance tests below, this is the mechanism used by the Synesis Atomic_ API and the WinSTL atomic functions.

10.3.3 Performance Comparisons

I've done a lot of talking about the various mechanisms available for atomic integer operations, so let's look at some facts. Before we do so, I'd like to stress that the figures in this section reflect the performance of Win32 operating systems on (single- and multi-) processor Intel architecture only. Other architectures and/or operating systems may have different characteristics.

I've examined seven strategies. For each there is a common global variable which is either incremented or decremented by the thread. The first strategy—Unguarded—does no locking, and simply increments or decrements the variable via the ++ or – operators. The next two use the architecture-dispatching techniques: the Synesis `Atomic_*` library functions and WinSTL's inline functions. The fourth calls the Win32 `Interlocked_*` system functions. The final three use a synchronization object—the Win32 CRITICAL_SECTION, the WinSTL `spin_mutex` and the Win32 mutex kernel object—to guard access to the critical region, within which the ++ or – operator is used to modify the variable. The results are shown in Table 10.2, which includes the total times for 10 million operations for 31 contending threads for each strategy. Since these were measured on different machines, the relative performance figures are also obtained.

Deliberately, the test program that exercised the various locking mechanisms spawned an odd number of threads, such that when all threads were complete the manipulated variables should have a large non-zero value equal to the number of iterations. This was a quick validation that the operations were indeed atomic. All of the cases shown, including the Unguarded manipulation on the uniprocessor machine behaved correctly. Verifying what we know about the nonatomic nature of some single instructions on multiprocessors, the Unguarded/SMP case

Table 10.2

Synchronization Scheme	Uniprocessor Machine		SMP machine	
	Absolute (ms)	% of unguarded	Absolute (ms)	% of unguarded
Unguarded ++ / −−	362	100%	525 (incorrect)	100%
Synesis Atomic_* API	509	141%	2464	469%
WinSTL atomic_* inline functions	510	141%	2464	469%
Win32 Interlocked* API	2324	642%	2491	474%
Win32 CRITICAL_SECTION	5568	1538%	188235	35854%
WinSTL spin_mutex	5837	1612%	3871	737%
Win32 MUTEX	57977	16016%	192870	36737%

produced wildly different values on each run, indicating that threads were interrupting each other's RMW cycles.

In terms of the performance, several things stand out. First, the relative cost of all mechanisms is higher on the multiprocessor machine, which indicates that multiprocessor caches are things that don't like being disturbed.

Second, as far as the architecture dispatching mechanism—the Synesis Atomic_ API and the WinSTL atomic_* inline functions—is concerned, it's fair to say that it does a very good job on the uni-processor system, being only 22% the cost of the Win32 `Interlocked_*` system library functions, and only being 141% the cost of the unguarded ++/−− operators. On the multiprocessor machine the additional cost over and above the LOCK for the processor test is very acceptable, being only an additional 1%. I would say that if you're writing applications that need to work on both single and multithreaded architectures, and you want to be able to ship a single version, you're likely to see significant benefits from using this dispatching technique.

The results show the well-known fact that mutexes, as kernel objects, represent a *very* costly way of implementing atomic operations, and you'd be crazy to use them for that purpose on Win32 systems.

Somewhat different from the mutex is the `CRITICAL_SECTION`. I don't know about you, but much of the wisdom gleaned as I was learning multithreaded programming on the Win32 platform advocated the use of the `CRITICAL_SECTION` as a much better alternative to the mutex. Indeed, it appears to be about 10 times as fast as the mutex on the uniprocessor system. However, it has about the same performance on the multiprocessor machine. Once again, you need to test your code on multiprocessor systems, in this case to verify your assumptions about the efficiency of the mechanisms you are using. I would say that the `CRITICAL_SECTION` is not a valid mechanism with which to get atomic operations; unlike the mutex I've actually seen a lot of use of it in clients' code bases.

You may wonder why anyone would use a spin mutex to implement atomic operations. Well, the atomic operations provided in Linux's `<asm/atomic.h>` are only guaranteed to provide a range of 24 bits. Furthermore, on some flavours of Linux, functions with the correct semantics—increment the value and return the previous value—are not available. By using the

conceptually simpler write/set functions the full-range atomic operations can still be provided, without incurring too high a performance penalty.

I hope these results give you some pause for thought in your implementations. Remember that this is Win32-specific; other architectures may have significantly different performance behaviors. But the main lesson is to profile, and to question your assumptions.

10.3.4 Atomic Integer Operations Coda

I've focused a great deal on atomic operations in this chapter, and I make no apologies for doing so. There are three reasons for this. First, they are something that I think could be incorporated directly into C++, which probably cannot be said for other multithreading and synchronization constructs, due to too-great differences or lack of broad availability.[11]

Second, they are an extremely useful construct, whose power is in inverse proportion to their simplicity. Using atomic integer operations only, one can achieve most or all the required thread safety in a significant number of C++ classes, as is shown in later chapters.

Last, the level of discussion of atomic operations in the literature is scant, to say the least. Hopefully by providing some focus here you'll think of them more often.

Atomic operations are available on many platforms, either as system library functions, or as nonstandard libraries, or by writing your own assembler versions. (I'm aware of the irony in promoting the use of assembler in a book largely dedicated to showing advanced C++ techniques; we live in a strange world.)

Even when we elect to use a (usually PTHREADS) mutex to implement our atomic operation, there are measures available to increase efficiency. The one thing to watch out for is not to use spin-mutexes in such cases. You'll be using several instances of a mutex class that are implemented in terms of an atomic integer API that is implemented on one, or a few, global mutexes. In such a case, you should use some preprocessor discrimination to make your chosen mutex a plain (PTHREADS-based) mutex, otherwise you'll be impacting negatively on performance, which won't be what you'll want or expect.

This is actually a good example of a broader truth to be found in multithreaded development. In practice we really need to consider the details of our synchronization needs and the facilities of the host system(s) on which our applications will run. It would be great if C++ did indeed stipulate atomic operations, but it's my personal opinion that providing standard higher-level synchronization primitives[12] and maintaining maximum efficiency over different architectures is a lot harder to do. As often the case, you are best served by being aware of all the multithreading tools at your disposal ([Bute1997, Rich1997]).

10.4 Multithreading Extensions

Now that we've looked at a few issues pertaining to multithreading, it may have occurred to you that it would be useful for the language to provide built-in support for multithreading operations. Indeed, several languages do provide multithreading constructs. The C++ tradition is to

[11]Having said that, there are some libraries, such as the excellent PThreads-Win32 (see Appendix A), which go some way to unifying the threading experience.

[12]There are moves afoot to make this happen in a future version of the standard, at which point I hope all the foregoing will be irrelevant. I doubt it, though.

favor the addition of new libraries rather than new language elements. We'll take a look at a couple of potential areas, see how the language might provide them, and how we can implement them using libraries (and a little bit of preprocessor trickery).

10.4.1 synchronized

D and Java have the `synchronized` keyword, which can be used to guard a critical region, as in:

```
Object obj = new Object();
. . .
synchronized(obj)
{
  . . . // critical code
}
```

One way to incorporate a `synchronized` keyword into the language would be to automatically translate the above code as follows:

```
{ __lock_scope__<Object> __lock__(obj);
{
  . . . // critical code
}
}
```

The `__lock_scope__` would be to all intents similar to the `lock_scope` template described in section 6.2. This would be pretty easy to do, and having an associated `std::lock_traits` template would enable an instance of any traits-able type to be synchronized in this way, which would not necessarily translate to a synchronization object lock.

This one is not a really strong contender for language extension, however, since with a modicum of macros we can achieve the same thing. Basically, all that is needed is the following two macros:

```
#define SYNCHRONIZED_BEGIN(T, v) \
  {                              \
  lock_scope<T> __lock__(v);
#define SYNCHRONIZED_END() \
  }
```

The only slight loss is that the type of the object would not be deduced for us, and also that the code looks somewhat less pretty:[13]

```
SYNCHRONIZED_BEGIN(Object, obj)
{
```

[13]It could be argued that the uglification is actually a benefit, since it increases the profile of the synchronized status of the critical region, which is a pretty important thing for anyone reading the code to take notice of.

```
  . . . // critical code
}
SYNCHRONIZED_END()
```

If you don't like the `SYNCHRONIZED_END()` part, you can always get a little bit trickier with your macro and define `SYNCHRONIZED()` macro as follows:

```
#define SYNCHRONIZED(T, v)                              \
  for(synchronized_lock<lock_scope<T> > __lock__(v);  \
      __lock__; __lock__.end_loop())
```

The `synchronized_lock<>` template class is only there to define a state[14] and to terminate the loop, since we can't declare a second condition variable within the `for` statement (see section 17.3). It is a bolt-in class (see Chapter 22) and looks like:

Listing 10.5

```
template <typename T>
struct synchronized_lock
  : public T
{
public:
  template <typename U>
  synchronized_lock(U &u)
    : T(u)
    , m_bEnded(false)
  {}
  operator bool () const
  {
    return !m_bEnded;
  }
  void end_loop()
  {
    m_bEnded = true;
  }
private:
  bool  m_bEnded;
};
```

There's another complication (of course!). As is described in section 17.3, compilers have different reactions to `for`-loop declarations, and if we were to have two synchronized regions in the same scope, some of the older ones would complain.

```
SYNCHRONIZED(Object, obj)
{
```

[14]It doesn't really define an `operator bool()`. We'll see why they're not used, and how to do them properly, in Chapter 24.

```
  . . . // critical code
}
  . . . // non-critical code
SYNCHRONIZED(Object, obj) // Error: "redefinition of __lock__"
{
  . . . // more critical code
}
```

Thus, a portable solution needs to ensure that each `__lock__` is distinct, so we have to get down and dirty with the preprocessor.[15]

```
#define concat__(x, y)            x ## y
#define concat_(x, y)             concat__(x, y)
#define SYNCHRONIZED(T, v)                          \
  for(synchronized_lock<lock_scope<T> >             \
      concat_(__lock__, __LINE__) (v);              \
      concat_(__lock__, __LINE__);                  \
      concat_(__lock__, __LINE__)  .end_loop())
```

It's ugly, but it works for all the compilers tested. If you don't need to be concerned with anachronistic `for` behavior, then just stick to the simpler version. The full versions of these macros and the classes are included on the CD.

10.4.2 Anonymous synchronized

There's a twist on the object-controlled critical region, which is that sometimes you don't have an object that you want to use as a lock. In this case, you can either just declare a static one in the local scope or, preferably, one in (anonymous) namespace scope in the same file as the critical region. You could also build on the techniques for the `SYNCHRONIZED()` macro, and produce a `SYNCHRONIZED_ANON()` macro that incorporates a local static, but then you run into a potential race condition whereby two or more threads might attempt to perform the one-time construction of the static object simultaneously. There are techniques to obviate this, as we'll see when we discuss statics in the next chapter, but it's best to avoid the issue. The namespace scope object is the best option in these cases.

10.4.3 atomic

Getting back to my favorite synchronization issue, atomic integer operations, one possible language extension would be to have an `atomic` keyword to support code such as the following:

```
atomic j = ++i; // Equivalent to j = atomic_preincrement(&i)
```

or, using the XOR exchange trick,[16]

```
atomic j ^= i ^= j ^= i; // Equiv. to j = atomic_write(&i, j);
```

[15] I'll leave it up to you to do a little research as to why the double concatenation is required.

[16] This is an old hacker's delight [Dewh2003], and frequent interview question. Test it out—it works, although I think it's not guaranteed to be portable!

It would be the compiler's responsibility to ensure that the code was translated into the appropriate atomic operation for the target architecture.[17] Unfortunately, the differences between processor instruction sets would mean that we'd either have to live with nonportable code, or that only a very few operations would be eligible for `atomic` decoration. We certainly would not want the compiler to use lightweight measures where it could and silently implement other operations by the locking and unlocking of a shared mutex: better to have these things expressly in the code as we do now.

It would be nice to have the `atomic` keyword for C and C++, for the limited subset of atomic integer operations that would be common to all architectures. However, using the `atomic_*` functions is not exactly a hardship, and it's certainly as readable—possibly more so—than the keyword form. Their only real downside is that they're not mandatory for all platforms; hopefully a future version of the C/C++ standard(s) will prescribe them.

10.5 Thread Specific Storage

All the discussion in the chapter has so far focused on the issues of synchronizing access to common resources from multiple threads. There is another side to threading, which is the provision of thread-specific resources or as it is more commonly known, *Thread-Specific Storage* (TSS) [Schm1997].

10.5.1 Re-entrancy

In single-threaded programs, the use of a local static object within a function is a reasonable way to make the function easier to use. The C standard library makes use of this technique in several of its functions, including `strtok()`, which tokenizes a string based one of a set of character delimiters:

```
char *strtok(char *str, const char *delimiterSet);
```

The function maintains internal static variables that maintain the current tokenization point, so that subsequent calls (passing `NULL` for `str`) return successive tokens from the string.

Unfortunately, when used in multithreaded processes, such functions represent a classic race-condition. One thread may initiate a new tokenization while another is midway through the process.

Unlike other race-conditions, the answer in this case is not to serialize access with a synchronization object. That would only stop one thread from modifying the internal tokenization structures while another was using them. The interruption of one thread's tokenization by another's would still occur.

What is required is not to serialize access to thread-global variables, but rather to provide thread-local variables.[18] This is the purpose of TSS.

---

[17]Note that I'm suggesting the keyword would apply to the operation, *not* the variable. Defining a variable atomic and then 50 lines down relying on that atomic behavior is hardly a win for maintainability. The clear intent, and grepability, of `atomic_*` functions is much preferable to that.

[18]And modern C and C++ run time libraries implement `strtok()` and similar functions using TSS.

10.5.2 Thread-Specific Data / Thread-Local Storage

Both the PTHREADS and Win32 threading infrastructures provide some degree of TSS. Just for consistency, the PTHREADS [Bute1997] version is called *Thread-Specific Data* (TSD), and the Win32 [Rich1997] version is called *Thread-Local Storage* (TLS), but it all amounts to the same thing.

They each provide the means to create a variable that will be able to contain different values in each thread in the process. In PTHREADS, the variable is known as a key; in Win32 an index. Win32 refers to the location for a key's value in each thread as a slot. I like to refer to keys, slots, and values.

PTHREADS' TSD works around the following four library functions:

```
int pthread_key_create(    pthread_key_t  *key
                       ,   void (*destructor)(void *));
int pthread_key_delete(    pthread_key_t  key);
void *pthread_getspecific( pthread_key_t  key);
int pthread_setspecific(   pthread_key_t  key
                       ,   const void     *value);
```

`pthread_key_create()` creates a key (of the opaque type) `pthread_key_t`. The caller can also pass in a cleanup function, which we'll talk about shortly. Values can be set and retrieved, on a thread-specific basis, by calling `pthread_setspecific()` and `pthread_getspecific()`. `pthread_key_delete()` is called to destroy a key when it is no longer needed.

Win32's TLS API has a similar quartet:

```
DWORD  TlsAlloc(void);
LPVOID TlsGetValue(DWORD dwTlsIndex);
BOOL   TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);
BOOL   TlsFree(DWORD dwTlsIndex);
```

The normal way in which these TSS APIs are used is to create a key within the main thread, prior to the activation of any other threads and store the key in a common area (either in a global variable, or returned via a function). All threads then manipulate their own copies of the TSS data by storing to and retrieving from their own slots.

Unfortunately, there are several inadequacies in these models, especially with the Win32 version.

First, the number of keys provided by the APIs is limited. PTHREADS guarantees that there will be at least 128; Win32 64.[19] In reality, one is unlikely to need to break this limit, but given the increasingly multicomponent nature of software it is by no means impossible.

The second problem is that the Win32 API does not provide any ability to clean up the slot when a thread exits. This means that one has to somehow intercept the thread's exit and clean up the resources associated with the value in that thread's slot. Naturally, for C++ folks, this is a

[19]Windows 95 and NT 4 provide 64. Later operating systems provide more (Windows 98/ME: 80, Windows 2000/XP: 1088), but code that must be able to execute on any Win32 system must assume 64.

painful loss of the automatic destruction that the language provides for us, and can be next to impossible to work around in some scenarios.

Despite PTHREADS providing a means for cleanup on thread termination, it still presents an incomplete mechanism for easy and correct resource handling. In essence, PTHREADS provides us with immutable RAII (see section 3.5.1). Although this is a great improvement on Win32's absence of any RAII, there are occasions when it would be desirable to be able to change the slot value for a given key. It's possible to manually clean up the previous values, but it'd be a lot better if that is done automatically for us.

The fourth problem is that PTHREADS assumes that the cleanup function is callable at the cleanup epoch. If, at the time that any of the threads exit, an API has been uninitialized, then it may no longer be valid to call a cleanup function that may call that API directly or indirectly. Similarly, and even more likely in practice, if a cleanup function is in a dynamic library, the cleanup function may no longer exist in the process's memory, which means it will crash.

10.5.3 __declspec(thread) and TLS

Before we look at handling those challenges, I'd like to describe one TSS mechanism that is provided by most compilers on the Win32 platform in order to ease the verbosity of using the Win32 TLS functions. The compilers allow you to use the `__declspec(thread)` qualifier on variable definitions, as in:

```
__declspec(thread) int  x;
```

Now `x` will be thread specific; each thread will get its own copy. The compiler places any such variables in a `.tls` section, and the linker coalesces all of these into one. When the operating system loads the process, it looks for the `.tls` section and creates a thread-specific block to hold them. Each time a thread is created a corresponding block is created for the thread.

Unfortunately, despite being extremely efficient [Wils2003d], there's a massive drawback to this that makes it only suitable for use in executables, and not in dynamic libraries. It can be used in dynamic libraries that are implicitly linked, and therefore loaded at process load time, since the operating system can allocate the thread-specific block for all link units loading at application load time. The problem is what happens when a dynamic library containing a `.tls` section is later explicitly loaded; the operating system is unable go back and increase the blocks for all the existing threads, so your library will fail to load.

I think it's best to avoid `__declspec(thread)` in any DLLs, even ones that you're *sure* will always be implicitly linked. In the modern component-based world, it's entirely possible that the DLL may be implicitly linked to a component that is explicitly loaded by an executable produced by another compiler, or in another language, and that does not already have your DLL loaded. Your DLL cannot be loaded, and therefore the component that depends on it cannot be loaded.

10.5.4 The Tss Library

Having been bitten too many times by the four problems associated with the TSS mechanisms of PTHREADS and Win32, I got on the ball and wrote a library that provides the functionality I needed. It consists of eight functions, and two helper classes. The main functions, which are compatible with C and C++, are shown in Listing 10.6:

Listing 10.6

```
// MLTssStr.h - functions are declared extern "C"
int     Tss_Init(void);    /* Failed if < 0. */
void    Tss_Uninit(void);
void    Tss_ThreadAttach(void);
void    Tss_ThreadDetach(void);
HTssKey Tss_CreateKey( void (*pfnClose)()
                     , void (*pfnClientConnect)()
                     , void (*pfnClientDisconnect)()
                     , Boolean bCloseOnAssign);
void    Tss_CloseKey(     HTssKey  hEntry);
void    Tss_SetSlotValue( HTssKey  hEntry
                        , void     *value
                        , void     **pPrevValue /* = NULL */);
void    *Tss_GetSlotValue(HTssKey  hEntry);
```

Like all good APIs it has `Init`/`Uninit`[20] methods (see Section 6.3), to ensure that the API is ready for any clients that need it. It also has two functions for attaching and detaching threads that I'll talk about in a moment.

Manipulating keys follows the convention in providing four functions. However, these functions offer more functionality. For providing cleanup at thread termination, the `Tss_CreateKey()` function provides the optional callback function `pfnClose`; specify `NULL` if you don't need it. If you want that cleanup function to be applied to slot values when they are overwritten, you specify `true` for the `bCloseOnAssign` parameter.

Preventing code from untimely disappearance is handled by the two optional callback function parameters `pfnClientConnect` and `pfnClientDisconnect`. These can be implemented to do whatever is appropriate to ensure that the function specified in `pfnClose` is in memory and callable when it is needed. In my use of the API I have had occasion to specify the `Init`/`Uninit` functions for other APIs, or to lock and unlock a dynamic library in memory, or a combination of the two, as necessary.

`Tss_CloseKey()` and `Tss_GetSlotValue()` have the expected semantics. `Tss_SetSlotValue()`, however, has an additional parameter, `pPrevValue`, over its PTHREADS/Win32 equivalents. If this parameter is `NULL`, then the previous value is overwritten, and subject to the cleanup as requested in the key creation. However, if this parameter is non-`NULL`, then any cleanup is skipped, and the previous value is returned to the caller. This allows a finer-grained control over the values, while providing the powerful cleanup semantics by default.

Being a C API, the natural step is to encapsulate it within scoping class(es), and there are two provided. The first is the `TssKey` class. It's not particularly remarkable—it just simplifies the interface and applies RAII to close the key—so I'll show only the public interface:

Listing 10.7

```
template <typename T>
class TssKey
```

[20] A little tip for all those who use British spelling out there: you can avoid pointless arguments about Initialise vs Initialize with your U.S. friends by using a contraction.

```
{
public:
  TssKey( void (*pfnClose)(T )
        , void (*pfnClientConnect)()
        , void (*pfnClientDisconnect)()
        , Boolean bCloseOnAssign = true);
  ~TssKey();
public:
  void  SetSlotValue(T value, T *pPrevValue = NULL);
  T     GetSlotValue() const;
private:
  . . . Members; hide copy ctor and assignment operator
};
```

The implementation contains static assertions (see section 1.4.7) to ensure that `sizeof(T) == sizeof(void*)`, to prevent any mistaken attempt to store large objects by value in the slot. The values are cast to the parameterizing type, to save you the effort in client code.

The next class is more interesting. If your use of the slot value were to create a single entity and then to reuse it, you'd normally follow the pattern in Listing 10.8:

Listing 10.8

```
Tss key_func(. . .);
. . .
OneThing const &func(Another *another)
{
  OneThing *thing = (OneThing*)key_func.GetSlotValue();
  if(NULL == thing)
  {
    thing = new OneThing(another);
    key_func.SetSlotValue(thing);
  }
  else
  {
    thing->Method(another);
  }
  return *thing;
}
```

However, if the function is more complex—and most are—then there may be several places where the slot value may be changed. Each one of these represents the possibility for a resource leak due to a premature return before the call to `SetSlotValue()`. For this reason the scoping class `TssSlotScope`, shown in Listing 10.9, is provided. I confess I have a perverse affection for this class, because it's a kind of inside-out RAII.

Listing 10.9

```
template <typename T>
class TssSlotScope
{
```

```
public:
  TssSlotScope(HTssKey hKey, T &value)
    : m_hKey(hKey)
    , m_valueRef(value)
    , m_prevValue((value_type)Tss_GetSlotValue(m_hKey))
  {
    m_valueRef = m_prevValue;
  }
  TssSlotScope(TssKey<T> &key, T &value);
  ~TssSlotScope()
  {
    if(m_valueRef != m_prevValue)
    {
      Tss_SetSlotValue(m_hKey, m_valueRef, NULL);
    }
  }
private:
  TssKey  m_key;
  T       &m_valueRef;
  T const m_prevValue;
// Not to be implemented
private:
  . . . Hide copy ctor and assignment operator
};
```

It is constructed from a TSS key (either `TssKey<T>`, or an `HTssKey`) and a reference to an external value variable. The constructor(s) then set the external variable to the slot's value, via a call to `Tss_GetSlotValue()`.

In the destructor, the value of the external variable is tested against the original value of the slot, and the slot's value is updated via `Tss_SetSlotValue()` if it has changed. Now we can write client code much more simply, and rely on RAII to update the thread's slot value if necessary.

Listing 10.10

```
OneThing const &func(Another *another)
{
  OneThing                *thing;
  TssSlotScope<OneThing*> scope(key_func, thing);

  if( . . . )
    thing = new OneThing(another);
  else if( . . . )
    thing = . . .;
  else
    . . .

  return *thing;
} // dtor of scope ensures Tss_SetSlotValue() is called
```

So we've seen how to use the Tss library, but how does it work? Well, I'm going to leave you to figure out the implementation,[21] but we do need to have a look at how the thread notifications are handled. This involves the two functions I've so far not described: `Tss_ThreadAttach()` and `Tss_ThreadDetach()`. These two functions should be called when a thread commences and terminates execution, respectively. If possible, you can hook into your operating system or run time library infrastructure to achieve this. If not, then you will need to do it manually.

On Win32, all DLLs export the entry point `DllMain()` [Rich1997], which receives notifications when the process loads/unloads, and when threads commence/terminate. In the Synesis Win32 libraries, the base DLL (MMCMNBAS.DLL) calls `Tss_ThreadAttach()` when its `DllMain()` receives the `DLL_THREAD_ATTACH` notification, and calls `Tss_ThreadDetach()` when it receives `Tss_ThreadDetach()`. Since it is a common DLL, all the other members of any executable can just use the Tss library, without being concerned with the underlying setup; it all just works.

Listing 10.11

```
BOOL WINAPI DllMain(HINSTANCE, DWORD reason, void *)
{
  switch(reason)
  {
    case DLL_PROCESS_ATTACH:
      Tss_Init();
      break;
    case DLL_THREAD_ATTACH:
      Tss_ThreadAttach();
      break;
    case DLL_THREAD_DETACH:
      Tss_ThreadDetach();
      break;
    case DLL_PROCESS_DETACH:
      Tss_Uninit();
      break;
  }
  . . .
```

On UNIX, the library calls `pthread_key_create()` from within `Tss_Init()` to create a private, unused key whose only purpose is to ensure that the library receives a callback when each thread terminates, which then calls `Tss_ThreadDetach()`. Since there is no mechanism for a per-thread initialization function in PTHREADS, the Tss library is written to act benignly when asked for data for a nonexistent slot, and to create a slot where one does not exist when asked to store a slot value. Thus, `Tss_ThreadAttach()` can be thought of as a

[21] Or to take a peek on the CD, since I've included the source for the library. Take care, though, it's old code, and not that pretty! It's probably not that optimal, either, so it should be taken as a guide to the technique, rather than the zenith of TSS library implementations.

mechanism for efficiently expanding all active keys in response to a thread's commencement, rather than doing it piecemeal during thread processing.

If you're not using PTHREADS or Win32, or you're not happy to locate your library in a Win32 DLL, you should ensure that all threads call the attach/detach functions. However, even if you cannot or will not do this, the library is written such that when the final call to `Tss_Uninit()` is received, it performs the registered cleanup for all outstanding slots for all keys.

This is a powerful catchall mechanism, and the only problem you'll have relying on this—apart from tardy cleanup, that is—is if your cleanup function must be called from within the same thread to deallocate the resource that was used to allocate it. If that's the case, and you can't ensure timely and comprehensive thread attach/detach notification, then you're out of luck. What do you want—we're only protoplasm and silicon!

10.5.5 TSS Performance

So far I've not mentioned performance. Naturally, the sophistication of the library, along with the fact that there is a mutex to serialize access to the store, means that it has a nontrivial cost when compared with, say, the Win32 TLS implementation, which is very fast indeed [Wils2003f]. In one sense, there's not an issue, since if you need this functionality, then you're going to have to pay for it somehow. Second, the cost of a thread switch is considerable, potentially thousands of cycles [Bulk1999], so some expense in TSS will probably be a lesser concern. However, we cannot dismiss the issue. Measures you can take to minimize the costs of the Tss library, or any TSS API, are to pass TSS data down through call chains, rather than have each layer retrieve it by itself and thereby risk precipitating context switches due to contention in the TSS infrastructure. Obviously this cannot be achieved with system library functions, or other very general or widely used functions, but is possible within your own application implementations.

Further, for the Tss store it's good to use the `TssSlotScope` template, since it will only attempt to update the slot when the value needs to be changed.

As usual, the choice of strategies is yours, representing a trade-off between performance, robustness, ease of programming, and required functionality.