Happiness Is An Option

Multithreading and a solution to the readers/writers problem

by Julian Bucknall

There's one thing about writing a monthly column: you get to describe some bad things that have happened to you, to give vent to your frustration, as it were, and your readers will forgive you your off-topic introduction. Providing, of course, that you also describe some snazzy algorithm.

So, this column is being written in Washington, DC. Wow, I can hear you say, lucky Julian, jetsetter *extraordinaire*, flying to international hot spots at the drop of the proverbial hat. The James Bond of algorithm-meisters. Ah, I wish it were that simple and glamorous.

I'm afraid the reality is rather more mundane. I had a flight from Denver to Paris, with connection in Washington, DC. I was going to Paris as a Christmas present from my wife, Donna, to see my favourite band, Pet Shop Boys, in concert. I'd missed their one and only concert in Denver, I was acting in a play at the time, and so Donna had saved up to send me to Paris to see them there. United Airlines had different ideas, as it happened. The first hop of my journey was delayed (something about a hydraulic line) and I arrived in Washington three minutes after the connecting flight to Paris had left. Nothing for it but to stay the night in Washington to await the next flight to Paris.

Which was at 6pm the next day.

So, I was going to spend one day of my three-day jaunt to *la belle France* at the Hilton Hotel and at Dulles airport. Thanks very much, United. Well, us connecting flight people were asked to pick up our luggage for the overnight stay. But they couldn't find *my* suitcase.

Well, excuse me while I *scream*. Anyway, here I am at Dulles (for which the word *dull* was invented) the next day. Five and a half hours to go. My trusty HP palmtop has a new set of batteries. No drugstore in sight, so by the time I get to Charles de Gaulle airport, the words fragrant and Julian would not appear in the same sentence unless you were being sarcastic.

Enough about my travails, otherwise Our Esteemed Editor will be wondering if I'm deliberately padding the article! This month we shall talk about multithreaded programming and, in particular, the readers/writers problem. The idea came from a message in one of TurboPower's newsgroups where the writer was complaining about a speed problem when one of SysTools' containers was used across multiple threads.

Some Terminology

First, we shall discuss a few concepts and review some functionality provided by Win32 systems for multithreaded programs. Then, I'll talk about the readers/writers problem and we'll see an interesting solution for it.

With Win32 operating systems, executing applications (*processes* in Windows-speak) can spawn off separate routines which can execute at the same time as the main program. These routines are called *threads*, and the main execution thread of the application is called the *primary thread*.

On single processor machines, the operating system will execute each of the threads for a short time in a round-robin fashion, according to their priority. In other words, one at a time, one after the other, very quickly, but to the programmer it is as if these threads execute concurrently. On multiprocessor machines, it is very likely that an application's threads *will* execute concurrently, one per processor.

This all sounds great, except when you consider what happens when threads have to share data. If we're not very careful, we get what's known as a *race condition*: corruption of data by two or more threads because they're updating it at the same time.

Consider the following scenario. There is a `longint` value, a counter, say, that is being read and incremented by several threads. The counter is being used to time stamp some data that is being written elsewhere. The algorithm is supposed to work like this: the routine reads the current value of the counter, increments it, uses the new value as the timestamp, and then updates the counter variable for the next time. Simple enough, so let's see how it breaks (and how badly) in a multithreaded app.

Thread A reads the current value of the counter. Before it has a chance to increment it and write the new value back to the variable, the thread is swapped out by the operating system and thread B starts running again. It also reads the counter in preparation for incrementing it. It does so, and then writes the new value back. It is then swapped out by the operating system and thread A starts up again. It now increments the value it read and writes the new value back to main memory. Bzzzt! The effect of all this is that threads A and B get the same timestamp, which certainly wasn't supposed to happen. This is the famous race condition: both threads are racing to complete their operation before they're swapped out.

Let me make a couple more points, without trying to beat this simple example to death. First, an obvious one: if the machine is a multiprocessor machine, the two threads are likely to be executing at the same time, and no swapping would take place. They can actually access the same data (that is,

the same memory location) at exactly the same time. The second one is subtler: there is no guarantee by the Pentium or the operating system that all of the four bytes contained in the `longint` counter will be updated at the same time. The reader of the counter may actually manage to read two bytes of the old value and two bytes of the new one, we cannot be sure. With the example of the counter, we may not even notice this effect, but if the `longint` value were a pointer instead, it could be fatal.

Of course, these things may not be noticed in testing: the inviolable Programmer's Law says that they only strike when the application is with your customers!

How do we combat the race condition? We use one (or more) of the Win32 synchronisation objects and the `WaitForSingleObject` or `WaitForMultiple` objects routines.

Critical Sections

We'll briefly discuss these objects and how they're used. However, before we do, there is a 'lightweight' synchronisation object that doesn't fit in with the rest of the family that we should talk about. This is the critical section, and it is usually the very first synchronisation object that programmers learn how to use.

The *critical section* object is so called because it was designed to mark a section of code (usually code that accesses some data or data structure) in a serialised fashion; it is critical that no two threads can execute the same code at the same time. The critical section object doesn't come with a lot of options: we can create one, acquire exclusive access to it (known as *entering* the critical section), release this exclusive access (*leaving* the critical section), and destroy it.

➤ *Listing 1: Simple removal of a race condition.*

```
var
  CounterCS : TRTLCriticalSection;
  GlobalCounter : integer;
function GetCounterSafely : integer;
begin
  EnterCriticalSection(CounterCS);
  Inc(GlobalCounter);
  Result := GlobalCounter;
  LeaveCriticalSection(CounterCS);
end;
```

For our simple example, we would write a routine that performs the read, increment and write operations. We would then insert the call to acquire the critical section object (created elsewhere) at the beginning of the routine (`EnterCriticalSection`) and a call to release it at the end (`LeaveCriticalSection`). Listing 1 shows this simple serialised access routine. We assume, in this routine, that the critical section already exists; presumably it was created at the start of the application (by calling `InitializeCriticalSection`) and will be destroyed at the end (with `DestroyCriticalSection`). In Delphi the critical section is a complex record structure of type `TRTLCriticalSection`, defined in the `Windows` unit.

If this routine is always used to access and update the counter, we can guarantee the race condition will not happen. Let's walk through the same execution scenario as before. Thread A runs the routine. It acquires the critical section and then reads the current value. At this point the operating system swaps threads and thread B starts up again. It calls the same routine. It first tries to acquire the critical section. It cannot (thread A has exclusive control of it) and so the operating system blocks the thread's execution until the critical section becomes free again. Thread B is forced to be swapped out and thread A starts up again. It completes its access to the counter and updates it with the new value. It then releases its exclusive ownership of the critical section. At some later point, the operating system will start up thread B again and it will get exclusive access to the critical section and read, increment and update the counter in a protected manner.

In Windows NT and Windows 2000 there is another API routine we could use: `TryEnterCriticalSection`. This returns immediately, either successfully entering the critical section and returning true, or noticing that some other thread has the critical section and returning false. No blocking occurs. We won't discuss that here, since it's not available for Windows 9x.

Deadlocks

Sounds simple enough. What are the problems? Well, there's only one in particular for such a simple example. Suppose the critical section object protected a piece of code that had an infinite loop (I would assume something not too obvious, because we don't deliberately write such things, do we?). The blocked thread or threads would never become unblocked: they would wait forever.

If we had several such protected routines then there is another problem, related to the first. Assume there are two critical sections in the application, X and Y. To perform a particular function, both critical sections are required. Thread A takes an execution path that acquires X first and then Y, whereas B acquires Y first and then X. Let A acquire critical section X, but before it can acquire critical section Y, the operating system does its magic and B starts running. It acquires critical section B and then goes on to acquire X. It fails (since A has it) and so the thread is blocked. Thread A starts up again and the first thing that happens is that it tries to acquire Y. It can't, B has it, therefore it is blocked as well. The upshot is that both threads are blocked waiting for each other to release a critical section. This is known as *deadlock*.

Deadlocks are often very difficult to diagnose and fix. In using critical sections, their effect is compounded because there is no timeout available. If there were, we could wait a certain time and then if we failed to acquire some critical section we could back out of our current processing, releasing other synchronisation objects in the process, and try again. Another method is to make sure we always acquire the synchronisation objects in the same order.

More Synchronisation

Having talked at length about critical sections, we should now discuss the other synchronisation

objects provided by Win32. The others are all characterised by having, and being referenced by, a handle, which can be used by the `WaitForSingleObject` or `WaitForMultipleObjects` API routines. `WaitForSingleObject` takes a single handle and then waits for that handle to become *signalled*. A handle, even something like a file handle, a process handle, or a thread handle, has two states: unsignalled and signalled. It's as if the handle had a little flag that is raised or lowered, raised meaning signalled and lowered meaning unsignalled. Depending on the object behind the handle, the handle becomes signalled in various ways, which we'll discuss for each object in a moment. `WaitForSingleObject` merely blocks the thread calling it until the referenced handle's flag goes up. `WaitForMultipleObjects`, as you might have guessed, accepts several handles and waits for one or all of them to become signalled (you select which it's to be).

One important point to realise about both `WaitFor...` routines is that they accept a timeout value. You can select to wait forever, or you can decide to wait for a certain short period of time only (you would get an error value back it the timeout period expired). Another point is the `WaitFor...` routines cannot work with critical sections: a critical section has no handle.

The first synchronisation object with a handle is the *mutex*. This object is very similar to the critical section, in that it can be acquired and released, and only one thread can acquire it at one time (mutex is short for *mut*ual *ex*clusion, which describes this functionality). When the mutex is owned by a thread it is unsignalled, when it is not owned by any thread it is signalled (it's as if the mutex is waving its flag saying 'I'm free!'). The `WaitForSingleObject` API routine sets the signalled flag, and the `ReleaseMutex` routine clears it.

The differences to critical sections are two-fold. Firstly, a mutex has a name and can be referenced from several processes (that is, applications), let alone from several threads. For example, with a mutex you can protect access to some shared memory between two or more programs. Secondly, using a mutex for a particular job takes longer than using a critical section. If you don't want the timeout facility, or you don't want to use it across different processes, or you don't want to wait for multiple objects, you should use a critical section: it's more efficient and faster. There's a third difference for advanced developers: the mutex has a set of security attributes associated with it, the critical section does not.

There are six API routines associated with a mutex. `CreateMutex` creates a new mutex (you can choose to own the new mutex from the start). `OpenMutex` returns a handle to an already existing mutex (although `CreateMutex` will do that for you if the mutex already exists). The two `WaitFor` routines acquire a mutex (or 'un-signal' it). `ReleaseMutex` releases the mutex, or signals it. And `CloseHandle` destroys the mutex.

The second Win32 synchronisation object in this category is the *semaphore*. This object is somewhat hard to explain, since it can be used in various bizarre ways. In one way, it's like a mutex that can be owned by several threads at once: you define the number when you create the semaphore. Let's assume the semaphore has an internal count of owners. If this value is greater than zero, the semaphore is signalled, if it is zero the semaphore is unsignalled. Every time one of the `WaitFor` routines succeeds this internal count is decremented. Eventually, the allowed number of threads own the semaphore and the count becomes zero. The semaphore becomes unsignalled. All subsequent `WaitFor` calls will block until one of the current owning threads calls `ReleaseSemaphore`. This routine increments the internal count of the semaphore, thereby signalling it. The operating system will release only one of the waiting threads, which will then decrement the internal count and un-signal the semaphore. Again, like the mutex, semaphores can be shared across several processes.

Lots of books at this point describe a mutex as a semaphore that has a maximum of one owner. This is misleading, in my view. The first thing to realise about the semaphore is that threads don't own one. A mutex is owned by a thread, a semaphore isn't. A semaphore is more of a counting object: it's a gateway through which threads have to pass to get to the other side. Only a certain number of threads can get through and, once the right number have passed through, the semaphore blocks all the others. It's a bit like a parking lot: only a certain number of cars can get in, after which you have to wait for someone else to leave before you can park.

Win32 provides six API routines to use a semaphore. `CreateSemaphore` creates a new semaphore and defines the maximum number of threads that can own the semaphore at once. `OpenSempahore` opens an existing semaphore (although `CreateSemaphore` can do that as well). The two `WaitFor` routines, if successful, will decrement the internal count by one. `ReleaseSemaphore` increments the internal count by one, signalling the semaphore in the process (this allows only one thread to be released and get in). Finally, `CloseHandle` destroys the semaphore.

The next Win32 synchronisation object is the *event*. This is where the nomenclature gets confusing for Delphi programmers: components using the VCL also have things called events, they're properties that define a routine to be executed. I therefore call the Windows event objects *Win32Events*, to separate them in my own mind from Delphi's events.

A Win32Event is an object that allows a thread to learn that something has happened, in other words, that an event has occurred. The assumption is that one thread is going to set something up and then tell one or more other threads that the event has occurred and that they can wake up and do something. The Win32Event remains unsignalled until the first

thread signals it. There are two possibilities for what happens then: either just one thread is released, or all the threads are released. In the first situation, the Win32Event is known as an *auto-reset event*: the Win32Event is signalled, one thread that has called a `WaitFor` routine on the object is released, and the Win32Event is immediately unsignalled again (the object is automatically reset to the unsignalled state). In the other scenario, the Win32Event is known as a *manual-reset event*. The Win32-Event is signalled, all threads waiting on it are released, and then (possibly) the Win32- Event is then reset to an unsignalled state by explicitly writing code to do it.

As an example, suppose there is a thread setting up a buffer. There are several threads that are going to be reading the data in this buffer (in other words, no writing). The reader threads all call a `WaitFor` routine on a Win32Event object. The thread that is writing data to the buffer finishes its write cycle and then signals the Win32Event to indicate that the buffer is ready to be used and be read from. All the waiting threads are released and start reading. The writing thread can then reset the Win32Event to unsignalled and start preparing the next buffer-full.

There are seven API routines that deal with Win32Events. `CreateEvent` creates a new Win32-Event, and there are parameters that define what kind of Win32-Event is being created and whether it is initially signalled. `OpenEvent` opens an existing Win32Event (although, again, the `Create` routine can do this). There are the usual two `WaitFor` routines. After that there are three routines that alter the signal status of the Win32Event: `SetEvent`, `ResetEvent` and `PulseEvent`. `SetEvent` sets a manual-reset Win32Event to signalled. `ResetEvent` sets one to unsignalled. `PulseEvent` can work with either manual-reset or auto-reset events: in the first case, it signals the Win32Event, releasing all waiting threads, and then resets the event to unsignalled. In the second case, the Win32Event is signalled, one thread is released, and the event is unsignalled.

Readers/Writers Problem

After this brief recap of basic multithreading and synchronisation objects, we'll take a look at the particular problem that prompted this column, at least in a much simplified form. Suppose we wish to share a `TList` amongst several threads. There will be two types of access: reading the information in the `TList` (for example, iterating through the contents), and adding, deleting, or modifying items in the `TList`. These can be characterised as reader threads and writer threads, or more succinctly as *readers* and *writers*. Note that I'm not saying that a particular thread is always going to be a reader or a writer, but in general at some point in time it will use the `TList` just for reading, a reader, whereas at another time it will update the `TList`, a writer. If you think about it, there can be many threads all reading the `TList` at the same time (they're not going to be updating the `TList`, after all), but only one thread can be updating the `TList` at any one time. If a writer thread gains control, it must complete its update before any reader threads, or the next writer thread, can start running. This is an example of a *readers/writers problem*.

How do we design a synchronisation strategy for this scenario? The simplest solution is to use a critical section. Indeed, this is what Delphi itself provides in the `TThreadedList` class, available in Delphi 3 and above. Essentially, the synchronisation strategy is implemented like this: every access to the `TList` is protected with a critical section or mutex. In Delphi's version, the `TThreadedList` provides a method called `LockList` that enters a critical section and returns the internal `TList`. The thread is then free to use this `TList` object until it has finished, at which point the thread routine is supposed to call `UnlockList` to leave the critical section.

Although this solution works, and works very well, it has a blindingly obvious drawback: only one thread can access the `TList` at any one time. There is no differentiation between read access (which doesn't alter the list) and write access (which does). As we saw, there could be many readers of the `TList` at any one time, the restriction is that there should be only one writer. This solution, although simple to implement, is big overkill. It does not enable us to make the most efficient use of the `TList`.

This, in essence, was the subject of the message on TurboPower's newsgroup. The container in question was protected in multi-threaded mode by this type of access strategy: only one thread can access the container at any one time, whether the thread just wants to read the data or update it.

Can we use any other synchronisation object? A mutex has the same effect as the critical section, so that can't help. The semaphore looks promising (we need several readers to run at the same time), but a moment's reflection will convince you that it's too primitive for what we want. The event also looks interesting, but again it fails as being too simple a synchronisation object. So, what can we do?

Well, we shall try and use two or more primitive synchronisation objects in some combination, in order to give us what we require. We shall design a compound synchronisation object.

Readers/Writers Solution

Let's define what we'd like the compound object to do. We need a single object that can be used by reader and writer threads to synchronise access to the `TList`. It should allow several reader threads to be active at once. It should allow only one writer thread to be active at any one time, and, if one is, no reader threads should be allowed either (they might access something in the `TList` that is in the middle of being updated).

Ideally, we should set up the following behaviour as well. If a thread wishes to write to the `TList`, it should be able to tell the object so. The object will then block any new reader threads from running, until all the current reader threads

have finished and the writer thread can continue. If there is no writer thread waiting, a reader thread should be allowed to access the `TList` without hindrance. If we're clever, we should allow several writer threads to become queued. All this means that, in essence, the object forces a cycle of many reader threads using the `TList`, followed by a single writer thread, followed by many reader threads, and so on.

A tall order? Maybe. Let's start slowly. It seems clear from the definition that there must be some synchronisation object that a writer thread can signal, once it has completed its update, to allow reader threads to run. Conversely, there must be a synchronisation object that the final reader thread of a set of reader threads can signal, once complete, in order to release a writer thread. The compound object that we're designing, then, requires at least four methods. A reader thread calls the first method in order to start reading (note that it may get blocked inside this routine, waiting for a writer thread to finish its work). Once a reader thread has completed, it needs to call another routine to exit its use of the synchronisation object and maybe release a writer thread. Similarly, there must be two such routines for a writer thread. Let's call these four routines `StartReading`, `StopReading`, `StartWriting`, and `StopWriting`.

It's fairly easy to describe how this might now work, the implementation is somewhat harder. `StartReading` has several jobs. It must first check to see if a writer is waiting. If there is at least one, it must start waiting on a synchronisation object of some sort, the most likely candidates being a semaphore or a Win32Event. If a writer is running, `StartReading` must do the same. If there is no writer running or waiting, `StartReading` registers the thread as a reader, and the thread can continue its work immediately.

In the `StopReading` method, the reader must work out if it is the last reader to be running. If it is, and a writer is waiting, it must release the writer by signalling the object the writer is waiting on. If there is no writer waiting, there can't be any readers waiting either (can you see why?) and so the method must leave the object in such a state that either a reader or writer thread could start immediately.

The `StartWriting` method does several things, too. If a writer thread is active, it waits on the synchronisation object that will be used to release the next writer. If there are one or more reader threads active, it does the same. Otherwise, it registers itself as writing and continues.

The `StopWriting` method deregisters the thread running it as a writer and then checks to see if one or more readers are ready to go. If so, it signals the synchronisation object that the readers are waiting on and finishes. If there are no readers, it then checks for a writer waiting. If so, it releases one writer, by signalling the object they're all waiting on and then terminates. If neither case applies, it leaves the

compound object in a state such that either a reader or a writer could start immediately.

OK, then. From this functional description we can extract various bits of information. One, we need a variable to hold the number of readers waiting. Two, we need a variable to hold the number of writers waiting. Three, we need a variable to hold the number of readers currently executing. Fourthly, we need a `Boolean` flag to say that a writer is executing. Finally, we need some synchronisation objects to wrap it all up. Notice that it seems we could use the number of executing readers as zero to mean that a writer has control, but in reality zero could mean that either a writer is active or no thread is active at all. So, we'd better stick to having the two special variables.

Since there are four interrelated variables, we need to wrap the calls to read and update them inside a critical section or a mutex. A mutex is better, since we could have a timeout on getting control. That's synchronisation object number one. Each of the four methods would acquire the mutex as a first step and release it as the final one. However, recall that the methods which allow the reader to start may block inside the routine. It would be an automatic deadlock should this block occur in between the code to acquire or release the controlling mutex, so we must make sure it occurs outside, after the mutex is released.

Since there can only be one writer active at once, it seems to make sense for the synchronisation object that serialises the writer threads to be a mutex as well, since a mutex can only be owned by one thread. In reality it is easier if we used a semaphore. The reason is simple: we don't actually want to *own* the synchronisation object, because there is no great place to release it. Indeed, you will see that we shall wait for a semaphore in one thread and release it from another. This is not possible with a mutex: the thread that acquires the mutex owns it.

The synchronisation object for the readers? Well either a semaphore or a manual-reset Win32-Event would make sense. Probably our best bet is to use a semaphore, the event object would have problems (it relies on all the threads waiting for the event object to be signalled, whereas in fact a thread could be in a state where it hadn't called the `WaitFor` routine yet).

```
type
  TaaReadWriteSync = class
    private
      FBlockedReaders : THandle; {a semaphore}
      FBlockedWriters : THandle; {a semaphore}
      FController     : THandle; {a mutex}
      FActiveReaders  : integer;
      FActiveWriter   : boolean;
      FWaitingReaders : integer;
      FWaitingWriters : integer;
    protected
    public
      constructor Create;
      destructor Destroy; override;
      procedure StartReading;
      procedure StartWriting;
      procedure StopReading;
      procedure StopWriting;
  end;
```

➤ *Listing 2: The TaaReadWriteSync synchronisation class.*

Code Discussion

Listing 2 shows the interface for the synchronisation class we're creating, the `TaaReadWriteSync` class. It's printed in this article just to show you the various private fields that we'll be using in the four main methods.

Listing 3 gives the code for the `StartReading` method. We first acquire the controlling mutex. After this point we have control of the values of the internal fields. If there is at least one writer waiting to have a go, or there is one currently executing, we increment the number of waiting readers, release the controlling mutex and then wait for the 'blocked readers' semaphore to become signalled. If there are no writers waiting or running, we increment the number of executing readers, and release the mutex. Once we exit this method, we've either been released from waiting for the semaphore, or we went straight through. Notice that in the second case we incremented the number of running readers, but in the first we did not. This looks like a bug in the making, but hold fire for a moment.

Listing 4, on the other hand, shows the `StopReading` method. We first acquire the controlling mutex, as usual. This thread wishes to stop its reading activities and so it decrements the executing readers count. If the resulting value is non-zero, there are other reader threads still active, and so we just release the controlling mutex and exit the routine. If, however, it was the last active reader, the count is now zero, and we need to release a waiting writer (if there is one). To do this we release the blocked writers semaphore; in other words, we increment the count by one, and the system will release one and only one blocked writer thread, immediately reducing the

➤ *Listing 3: The StartReading method.*

```
procedure TaaReadWriteSync.StartReading;
var
  HaveToWait : boolean;
begin
  WaitForSingleObject(FController, INFINITE);  {acquire the controlling mutex}
  {if there is a writer executing or there is at least one writer
   waiting, add ourselves as a waiting reader, make sure we wait}
  if FActiveWriter or (FWaitingWriters <> 0) then begin
    inc(FWaitingReaders);
    HaveToWait := true;
  end else begin
    {otherwise add ourselves as another executing reader, make sure we don't wait}
    inc(FActiveReaders);
    HaveToWait := false;
  end;
  ReleaseMutex(FController);    {release the controlling mutex}
  if HaveToWait then
    WaitForSingleObject(FBlockedReaders, INFINITE);  {if we have to wait, do so}
end;
```

```
procedure TaaReadWriteSync.StopReading;
begin
  WaitForSingleObject(FController, INFINITE);  {acquire the controlling mutex}
  dec(FActiveReaders);                         {we're finishing reading}
  {if we are the last reader in this cycle and there is at least one writer
   waiting, release it}
  if (FActiveReaders = 0) and (FWaitingWriters <> 0) then begin
    dec(FWaitingWriters);
    FActiveWriter := true;
    ReleaseSemaphore(FBlockedWriters, 1, nil);
  end;
  ReleaseMutex(FController);                   {release the controlling mutex}
end;
```

➤ *Listing 4: The StopReading method.*

```
procedure TaaReadWriteSync.StartWriting;
var HaveToWait : boolean;
begin
  WaitForSingleObject(FController, INFINITE);  {acquire the controlling mutex}
  {if there are readers or another writer running, add ourselves as a
   waiting writer, and make sure we wait}
  if FActiveWriter or (FActiveReaders <> 0) then begin
    inc(FWaitingWriters);
    HaveToWait := true;
  end else begin
    {otherwise add ourselves as another executing writer, make sure we don't wait}
    FActiveWriter := true;
    HaveToWait := false;
  end;
  ReleaseMutex(FController);                   {release the controlling mutex}
  if HaveToWait then
    WaitForSingleObject(FBlockedWriters, INFINITE);  {if we have to wait, do so}
end;
```

➤ *Listing 5: The StartWriting method.*

count back to zero again, making sure that all the other writer threads remain blocked. Prior to that, though, the `StopReading` method decrements the number of waiting writers, and increments the number of running writers. The controlling mutex is then released. The overall effect of this is that a writing thread is released and the two counts for the writers are adjusted.

The `StartWriting` method is shown in Listing 5 and proceeds as follows. First thing, again, is to acquire the controlling mutex. If there are any running readers or writers, we increment the number of waiting writers, release the controlling mutex and then wait for the 'blocked writers' semaphore to be released. If the are no other running threads then we can start straight away. We increment the number of executing writers, release the controlling mutex, and exit the routine. Either way, once we exit the routine the number of active writers is set to one, either by the method itself, or by the `StopReading` method (clever, huh?).

Finally, Listing 6 shows the `StopWriting` method. First off: acquire the controlling mutex, of course. Then, as we are stopping writing, we decrement the number of active writers. We now check the number of waiting readers. If it is greater than zero we need to release them all. We enter a loop that decrements the number of waiting readers, increments the number of active readers, and releases the semaphore. This will in turn release one reader from waiting. Eventually at the end of the loop, all reader threads will have been released, and they are all active. If, on the other hand, there are no readers waiting, the method checks for any writers waiting. If there are, it releases one in the manner already described in `StopReading`. Finally, no matter what, it releases the controlling mutex.

The rest of this reader/writer synchronisation object can be found on this month's disk, not that it consists of much. The `Create` constructor allocates the two semaphores and the controlling mutex, the `Destroy` destructor closes their handles.

I hope you enjoyed this foray into multithreading in Win32. The readers/writers synchronisation object we wrote is useful in many situations and certainly should be part of your multithreaded algorithm arsenal. Next time we shall look at some more standard compound synchronisation objects and also look at monitors, which are yet another multithreaded programming construct (and also a bad pun, if ever I heard one). Later on this year, I'll be updating both columns to reflect the situation with Kylix and Linux. Stay tuned!

Julian Bucknall cut his teeth writing on Delphi topics with a multithreading chapter in a now defunct Delphi book. Just to let you know, this column was completed on the flight to Paris; he arrived there on the same day as the concert, which was fabulous. He's still hard at work writing the algorithms book, but responds to mail at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.

➤ *Listing 6: The StopWriting method.*

```
procedure TaaReadWriteSync.StopWriting;
var i : integer;
begin
  WaitForSingleObject(FController, INFINITE);  {acquire the controlling mutex}
  FActiveWriter := false;                      {we're finishing writing}
  {if there is at least one reader waiting, release them all}
  if (FWaitingReaders <> 0) then begin
    for i := pred(FWaitingReaders) downto 0 do begin
      dec(FWaitingReaders);
      inc(FActiveReaders);
      ReleaseSemaphore(FBlockedReaders, 1, nil);
    end;
  end else if (FWaitingWriters <> 0) then begin
    {otherwise, if there is at least one waiting writer, release one}
    dec(FWaitingWriters);
    FActiveWriter := true;
    ReleaseSemaphore(FBlockedWriters, 1, nil);
  end;
  ReleaseMutex(FController);  {release the controlling mutex}
end;
```