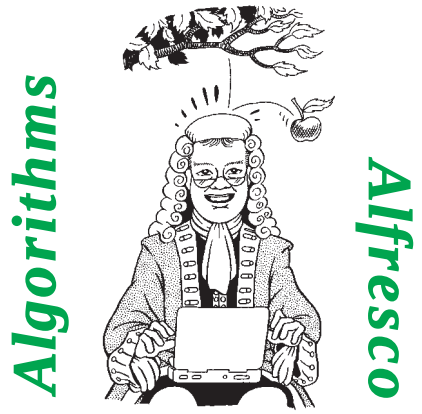
Hyperactive

More multithreading, this time with producers and consumers



by Julian Bucknall

At about the time I was writing last month's article, one of my dear friends, Jim Modrall, died suddenly of an aortic aneurysm. He was a mere 53 years old, a gentle, levelheaded man. I had acted with him in several plays over the past few years (happy times) and he was well known and liked within the Colorado Springs theatre community. He and his wife Nancy had always supported my wife Donna and I in our lives and careers (he used to be a District Attorney, Donna's chosen career, and would regale us with stories about some of his trials). With your permission, I would like to dedicate this article to his memory and to Nancy.

Change His Ways

Last month, in *Algorithms Alfresco*, we briefly reviewed the various multithreaded synchronization objects available in 32-bit Windows and then designed and implemented a compound synchronization object that solved the *Readers/Writers* problem. To recap, the new object enabled access to some data in order that multiple threads, which were either exclusively reading or updating the structure, were properly synchronized. We allowed many reader threads to access the data at the same time (since they were doing nothing to alter the structure), but restricted access so that a writer thread was the only thread accessing the structure at any particular time. The object we created, in essence, sequenced access so that many reader threads were active, followed by one writer thread, followed by many reader threads, and so on.

If you recall, we used a `TList` as the example data structure to which we wanted to synchronize access. If we were scrupulously honest and vigilant about using a `TaaReadWriteSync` object, we could access the `TList` properly without fear of corrupting data. But, as soon as we forget to use the class, even in only one thread, in any one part of the code, data would be corrupted sooner or later. We may not even notice straight away.

In Delphi 3 and above, Borland provided the `TThreadedList` class to try and avoid this problem. When a thread wants to use the `TList` encapsulated by the `TThreadedList`, it calls the `LockList` method. This returns a `TList` instance that the thread can then use with impunity until it calls `UnlockList` to release its exclusive access. (Internally the `TThreadedList` is using a critical section to lock access.) There is no way (unless we want to play silly coding tricks) to get hold of the internal `TList` unless we call the `LockList` method. This sounds pretty good, and we could certainly do much the same thing with last month's `TaaReadWriteSync` class.

However, it's still pretty long-winded and error-prone. We have to call the 'lock' method. We can then access the list. Afterwards we have to remember to call the 'unlock' method. Even worse, the 'lock' method returns a pointer to the internal list that we could try to use even *after* we'd called the 'unlock' method. Brrrr.

It would be much better surely if we had a class that encapsulated *all* access to the internal list, and that ensured that the required calls to lock and unlock the internal data structure were made. We then wouldn't *have* to worry about making sure we used the locking mechanism: it would be enforced on us by the data structure itself. We also wouldn't have access to the internal list since we would never get a pointer to it: it remains completely hidden.

Looking For Clues

In the multithreading community, this kind of object is known as a *monitor*. I don't know why the word 'monitor' was coined for such an object; it seems a little bizarre. Anyway, a monitor is a black box that encapsulates a resource or several resources. Inside the black box is a synchronization object to regulate proper access to the data by several threads and, of course, the data itself. Every method of the monitor ensures that the thread calling it is allowed to access the data. They each lock the object, perform the required access to the data, and then unlock the object. To the outside world, these shenanigans are completely hidden: the thread merely calls the `Add` method, or the `Count` property, and so on.

Let's encapsulate the readers/writers synchronization object into a thread-safe `TList` as a monitor. The first thing that we need to do is to create the class interface, and this must look the same as the `TList` interface. Listing 1 shows the class definition for the `TaaThreadedList`, where I have exposed the usual `TList` methods and properties. However, notice that the `read` clauses for each property define functions. In the 'real' `TList`, for example, the `read` clause for the `Count` property is merely the internal field, `FCount`, whereas in the `TaaThreadedList` it is defined as a call to a function. Sharing a data field in a multithreaded application means that we have to make it thread-safe and these functions, as we shall see, will enforce that for us.

Next, we need to categorize each method as a method that merely reads the TList or as a method that alters the structure or sequence of the TList; in other words, as reading methods or as writing methods. For the TList, it's fairly simple. The tlGetItem, tlGetCount, and tlGetCapacity methods are reading methods, for example; a lot of the others update the list one way or another.

Generally, the implementation of each method is simplicity itself, and trivial to code. We call the tlStartReading or tlStartWriting method to lock the list, call the required method of the internal TList to do the job, and then call the tlStopReading or tlStopWriting method to complete the job. Obviously we use try...finally blocks to ensure that the object is left in an unlocked state at the end of each method. I have shown a few representative methods of the TaaThreadedList in Listing 2. Interested readers can of course find the entire object source code on this month's companion disk.

➤ *Listing 2: Some TaaThreadedList methods.*

```
function TaaThreadedList.Add(aItem : pointer) : integer;
begin
  {Add is a writer}
  FLock.StartWriting;
  try
    Result := FList.Add(aItem);
  finally
    FLock.StopWriting;
  end;
end;
procedure TaaThreadedList.Delete(aInx : integer);
begin
  {Delete is a writer}
  FLock.StartWriting;
  try
    FList.Delete(aInx);
  finally
    FLock.StopWriting;
  end;
end;
function TaaThreadedList.IndexOf(aItem : pointer) : integer;
begin
  {IndexOf is a reader}
  FLock.StartReading;
  try
    Result := FList.IndexOf(aItem);
  finally
    FLock.StopReading;
  end;
end;
function TaaThreadedList.tlGetItem(aInx : integer) :
  pointer;
begin
  {tlGetItem is a reader}
  FLock.StartReading;
  try
    Result := FList[aInx];
  finally
    FLock.StopReading;
  end;
end;
```

```
type
  TaaThreadedList = class
    private
      FList : TList;
      FLock : TaaReadWriteSync;
    protected
      function tlGetItem(aInx : integer) : pointer;
      function tlGetCapacity : integer;
      function tlGetCount : integer;
      procedure tlPutItem(aInx : integer; aItem : pointer);
      procedure tlSetCapacity(aNewCapacity : integer);
      procedure tlSetCount(aNewCount : integer);
    public
      constructor Create;
      destructor Destroy; override;
      function Add(aItem : pointer) : integer;
      procedure Clear; virtual;
      procedure Delete(aInx : integer);
      procedure Exchange(aInx1, aInx2 : integer);
      function Extract(aItem : pointer) : pointer;
      function First : pointer;
      function IndexOf(aItem : pointer) : integer;
      procedure Insert(aInx : integer; aItem : pointer);
      function Last : pointer;
      procedure Move(aCurInx, aNewInx : integer);
      function Remove(aItem : pointer) : integer;
      property Capacity : integer
        read tlGetCapacity write tlSetCapacity;
      property Count : integer read tlGetCount write tlSetCount;
      property Items[aInx : integer] : pointer
        read tlGetItem write tlPutItem; default;
  end;
```

➤ *Listing 1: The TaaThreadedList class definition.*

Get It Through Your Heart

Now that we've extensively analysed a particular multithreaded problem, the readers/writers problem, and provided a standalone synchronization class for it, as well as a TList monitor, let's consider some more classic multithreaded problems and their solutions.

The readers/writers problem is a variant of a class of scenarios known as producer/consumer problems. In a producer/consumer problem, we have the situation where one or more threads are *producing* data that one or more threads are then using or *consuming*.

This definition is a little opaque, so let's consider a couple of examples. The first is video streaming. In this situation, we have a program that is displaying a video on the screen. Concurrently to us watching the movie, the program is downloading the video sequences from the internet as a stream of data. If this were implemented as a single-threaded program, we'd get a very choppy video. The program would download a chunk of the video, then display it, then get the next chunk, and so on. As a multithreaded application, however, the program could start up a thread to get the video data off the internet (or, indeed, several threads) and start a thread that just displays data. The thread or threads that download the data are the *producers* and the thread that displays the video is the *consumer*. We must of course synchronize the data access so that the consumer doesn't read a buffer's worth of data until the producer has actually finished downloading it. Similarly a producer cannot reuse a buffer until the consumer has displayed it all.

Another example is a stream copy routine. Usually, when we write a stream copy routine, we write a simple loop of reading a block of data from the source stream and then writing it to the destination stream. We alternate between reading data and writing it, each time waiting for the stream access to finish before moving to the next. If the reading code (the producer) were being executed in a different thread than the writing code (the consumer), we could perform both actions concurrently. Of course, we would have to synchronize the producer and consumer so that one wasn't using a buffer at the same time as the other.

Casting A Spell

Let's consider this second example and flesh it out. If we shared one buffer between producer and consumer, we wouldn't be that much better off than the original sequential stream copy method. The producer would have to fill the buffer, release the consumer to read the buffer, and then go to sleep. Once the consumer has read the buffer, it would re-awaken the producer to refill the buffer and then go to sleep itself. We'd get a series of read a buffer, write a buffer, but instead of being sequential in a single thread, we'd be continually switching from one thread to another; the threads are said to be in lockstep. In fact, due to the overhead of swapping between threads, it would run more slowly than the simple sequential case.

No, this situation cries out for a more sophisticated algorithm. We shall use several buffers. The producer will fill a buffer and put it in a queue, and then go ahead and fill another buffer. The consumer will get the buffer at the top of the queue and process it, before going back to get the next buffer. In this way the producer can go ahead and try and fill up the buffers in the queue as fast as it can and the consumer can, in turn, try and read them as fast as it can. The producer will stall if the consumer isn't fast enough and the queue fills up with unread buffers. Similarly the consumer will stall if the producer is slow and the queue empties.

In practice, the queue is implemented as a circular queue. The queue is created with all of its buffers pre-allocated. (We do *not* want to use the heap manager during the stream copy process since a critical section protects the heap manager in a multithreaded program. If we start to call memory allocation and deallocation routines from our threads they will block each other too easily.) The producer will fill up the buffer at the tail of the queue and advance the tail pointer. The consumer, on the other hand, will read the data in the buffer at the head of the queue and advance the head. The fill and the read processes can occur at the same time since they use different buffers.

If this were all we had to worry about, the code would be simple to write indeed: a single critical section to protect the reading and updating of the queue pointers and we'd be off. However, there are

```
TQueuedBuffers = class
  private
    FBufCount   : integer;
    FBuffers    : PBufferArray;
    FHead       : integer;
    FIsNotEmpty : THandle;
    FIsNotFull  : THandle;
    FTail       : integer;
  protected
    function qbGetHead : PBuffer;
    function qbGetTail : PBuffer;
  public
    constructor Create(aBufferCount : integer);
    destructor Destroy; override;
    procedure AdvanceHead;
    procedure AdvanceTail;
    property Head : PBuffer read qbGetHead;
    property Tail : PBuffer read qbGetTail;
    property IsNotEmpty : THandle read FIsNotEmpty;
    property IsNotFull : THandle read FIsNotFull;
  end;
constructor TQueuedBuffers.Create(aBufferCount : integer);
var
  i : integer;
begin
  inherited Create;
  {allocate the buffers}
  FBuffers := AllocMem(aBufferCount * sizeof(pointer));
  for i := 0 to pred(aBufferCount) do
    GetMem(FBuffers^[i], sizeof(TBuffer));
  FBufCount := aBufferCount;
  {create the semaphores}
  FIsNotFull :=
    CreateSemaphore(nil, aBufferCount, aBufferCount, '');
  FIsNotEmpty :=
    CreateSemaphore(nil, 0, aBufferCount, '');
end;
destructor TQueuedBuffers.Destroy;
var
  i : integer;
begin
  {destroy the semaphores}
  if (FIsNotFull <> 0) then
    CloseHandle(FIsNotFull);
  if (FIsNotEmpty <> 0) then
    CloseHandle(FIsNotEmpty);
  {free the buffers}
  if (FBuffers <> nil) then begin
    for i := 0 to pred(FBufCount) do
      if (FBuffers^[i] <> nil) then
        FreeMem(FBuffers^[i], sizeof(TBuffer));
    FreeMem(FBuffers, FBufCount * sizeof(pointer));
  end;
  inherited Destroy;
end;
procedure TQueuedBuffers.AdvanceHead;
begin
  inc(FHead);
  if (FHead = FBufCount) then
    FHead := 0;
end;
procedure TQueuedBuffers.AdvanceTail;
begin
  inc(FTail);
  if (FTail = FBufCount) then
    FTail := 0;
end;
function TQueuedBuffers.qbGetHead : PBuffer;
begin
  Result := FBuffers^[FHead];
end;
function TQueuedBuffers.qbGetTail : PBuffer;
begin
  Result := FBuffers^[FTail];
end;
```

➤ *Listing 3: The TQueuedBuffers class for one consumer.*

also two boundary conditions we have to worry about. If the queue is full (in other words, all the buffers in the queue are filled, waiting for the consumer to read them), the producer has nowhere to write a buffer's worth of data. It must therefore be blocked until the consumer has released at least one buffer to be reused. Similarly, if the queue is empty, the consumer must be blocked until such time that the producer has managed to fill another buffer.

For each of these two situations, we shall use a semaphore. The first semaphore will be the 'queue is not full' semaphore, the second will be the 'queue is not empty' semaphore. If the queue is not full, in other words the producer can fill at least one buffer (the one at the tail of the queue), the first semaphore is signalled.

Since it is the consumer that determines whether the queue is full or not, it is the consumer that will signal the semaphore: the producer will merely wait on it. If, on the other hand, the queue is not empty (that is, at least one buffer is ready to be read from the head of the queue), the second semaphore is signalled. This is the producer's job, since it is the producer that determines this situation.

Let's now write the stream copy routine. The routine accepts two parameters: the input stream and the output stream. It creates a special object of type `TQueuedBuffers`. This object contains all the resources necessary to implement a queued set of buffers: the buffers themselves, a semaphore for the 'queue is not empty' condition, and a semaphore for the 'queue is not full' condition. The latter semaphore is created with the full count already set (indicating that all of the buffers are unused). Listing 3 has the code for this simple class.

The copy routine then creates the two threads that will perform the copy, and resumes them (they are created suspended). The producer thread enters a loop. Each time through the loop, it waits on the 'queue is not full' semaphore, and when it is signalled, it reads a block of data from the source stream into the buffer at the head of the queue. It then advances the head pointer. Finally it signals the 'queue is not empty' semaphore and goes round the loop again.

The consumer thread's loop, on the other hand, proceeds as follows. First, it waits on the 'queue is not empty' semaphore. When signalled, it reads the buffer from the tail pointer and writes it to the destination stream. It then advances the tail pointer. Finally, it signals the 'queue is not full' semaphore and goes round the loop again. In order to signal that the producer has read all of the data from the source stream, each buffer in the queue has a count of bytes in the buffer. When zero, the consumer is to take this as meaning that the source stream was exhausted.

Listing 4 has the complete code for this routine and the two thread classes.

Some of you might have got a little worried after perusing the code and the description, pointing out that we were allowing multiple threads to update the same data at the same time: the queue's pointers. In fact, what happens is that the two semaphores provide us with enough synchronization that we cannot corrupt the queue. The head pointer is *only* read and updated by the producer thread; the tail pointer *only* by the consumer thread. Never do we compare head and tail pointers in one single thread (the usual way of determining whether a queue is

empty, or full, or somewhere in between). Instead we make use of the fact that a semaphore is a thread-safe counter. We preload the counter of the 'queue is not full' semaphore with the number of buffers available. After that point, it's as if we are passing tokens from the producer thread to the consumer thread via the two semaphores. The producer gets a token from the 'queue is not full' semaphore (decrementing its count) and then, after filling the head buffer, passes it to the 'queue is not empty' semaphore (incrementing its count). The consumer gets a token from this latter semaphore (decrementing its count) and then, after reading the tail buffer, passes it to the 'queue is not full' semaphore (incrementing its count).

Addicted To Love

Another producer/consumer problem that deserves special attention is the 'one producer, several consumers' scenario. There are two variants of this problem. The first is the one where the producer thread is producing a series of packets of data that the consumer threads pick off one by one and process. It doesn't matter which consumer processes which packet, all consumers are the same and the packets do not have to be serialized in any way. The other situation is where the producer thread is generating data that each of the consumer threads has to read.

This second situation is the more interesting. An example scenario is a stock quotes reader, a communications program that logs onto some private server and then just receives a stream of stock quote data. If we were sophisticated designers, we would implement this as a multithreaded program. The first thread would be the producer thread that receives the stream of data, chops it up into packets that are then placed on a queue. We would then add to our design several consumer threads that all read the same data. The first would get the data and display it continuously on the screen for the trader (a kind of tickertape display). The second thread would get the data and update a database containing our stock portfolio. The third would log the data in some form. There could be other threads doing other specific jobs, but as you can see each consumer thread ends up doing one particular job with the data, with all of them reading the same data.

This looks pretty difficult to synchronize, but if we take it gently, it's not so bad. We shall make things easier for ourselves in this article by describing a routine that copies a single stream to several different streams simultaneously. That way, we won't get bogged down with display issues and the like.

As with the multithreaded, multibuffered stream copy routine using a single producer and a single consumer, we shall use a circular queue of many buffers. This way the producer can continue to fill buffers at the tail of the queue, whilst the many consumers are all trying to read the buffers at the head of the queue. Since we now have many consumers we shall, in effect, have many queue heads, one per consumer. That way, a particular consumer can, if it doesn't

➤ *Listing 4: The producer thread, the consumer thread, and the stream copy routine.*

```
constructor TProducer.Create(aStream  : TStream;
  aBuffers : TQueuedBuffers);
begin
  inherited Create(true);
  FStream := aStream;
  FBuffers := aBuffers;
end;
procedure TProducer.Execute;
var
  Tail : PBuffer;
begin
  {do until the stream is exhausted...}
  repeat
    {get the 'queue is not full' semaphore}
    WaitForSingleObject(FBuffers.IsNotFull, INFINITE);
    {read a block from the stream into the tail buffer}
    Tail := FBuffers.Tail;
    Tail^.bCount := FStream.Read(Tail^.bBlock, BufferSize);
    {advance the tail pointer}
    FBuffers.AdvanceTail;
    {as we've written a new buffer, signal the
    'queue is not empty' semaphore}
    ReleaseSemaphore(FBuffers.IsNotEmpty, 1, nil);
  until (Tail^.bCount = 0);
end;
constructor TConsumer.Create(aStream  : TStream;
  aBuffers : TQueuedBuffers);
begin
  inherited Create(true);
  FStream := aStream;
  FBuffers := aBuffers;
end;
procedure TConsumer.Execute;
var
  Head : PBuffer;
begin
  {get the 'queue is not empty' semaphore}
  WaitForSingleObject(FBuffers.IsNotEmpty, INFINITE);
  {get the head buffer}
  Head := FBuffers.Head;
  {while the head buffer is not empty...}
  while (Head^.bCount <> 0) do begin
    {write a block from the head buffer into the stream}
    FStream.Write(Head^.bBlock, Head^.bCount);
    {advance the head pointer}
    FBuffers.AdvanceHead;
    {as we've read a buffer, signal the
     'queue is not full' semaphore}
    ReleaseSemaphore(FBuffers.IsNotFull, 1, nil);
    {get the 'queue is not empty' semaphore}
    WaitForSingleObject(FBuffers.IsNotEmpty, INFINITE);
    {get the head buffer}
    Head := FBuffers.Head;
  end;
end;
procedure AAThreadedcopyStream(
  aSrcStream, aDestStream : TStream);
var
  Buffers  : TQueuedBuffers;
  Producer : TProducer;
  Consumer : TConsumer;
  WaitArray : array [0..1] of THandle;
begin
  Buffers := nil;
  Producer := nil;
  Consumer := nil;
  try
    {create the queued buffer object (20 buffers)
     and the two threads}
    Buffers := TQueuedBuffers.Create(20);
    Producer := TProducer.Create(aSrcStream, Buffers);
    Consumer := TConsumer.Create(aDestStream, Buffers);
    {save the thread handles so we can wait on them}
    WaitArray[0] := Producer.Handle;
    WaitArray[1] := Consumer.Handle;
    {start the threads up}
    Consumer.Resume;
    Producer.Resume;
    {wait for the threads to finish}
    WaitForMultipleObjects(2, @WaitArray, true, INFINITE);
  finally
    Producer.Free;
    Consumer.Free;
    Buffers.Free;
  end;
end;
```

```
constructor TQueuedBuffers.Create(aBufferCount : integer;
  aConsumerCount : integer);
var
  i : integer;
begin
  inherited Create;
  {allocate the buffers}
  FBuffers := AllocMem(aBufferCount * sizeof(pointer));
  for i := 0 to pred(aBufferCount) do
    GetMem(FBuffers^[i], sizeof(TBuffer));
  FBufCount := aBufferCount;
  {create the semaphores}
  FConsumerCount := aConsumerCount;
  FIsNotFull :=
    CreateSemaphore(nil, aBufferCount, aBufferCount, '');
  for i := 0 to pred(aConsumerCount) do
    FIsNotEmpty[i] :=
      CreateSemaphore(nil, 0, aBufferCount, '');
end;
destructor TQueuedBuffers.Destroy;
var
  i : integer;
begin
  {destroy the semaphores}
  if (FIsNotFull <> 0) then
    CloseHandle(FIsNotFull);
  for i := 0 to pred(ConsumerCount) do
    if (FIsNotEmpty[i] <> 0) then
      CloseHandle(FIsNotEmpty[i]);
  {free the buffers}
  if (FBuffers <> nil) then begin
    for i := 0 to pred(FBufCount) do
      if (FBuffers^[i] <> nil) then
        FreeMem(FBuffers^[i], sizeof(TBuffer));
    FreeMem(FBuffers, FBufCount * sizeof(pointer));
  end;
  inherited Destroy;
end;
procedure TQueuedBuffers.AdvanceHead(aConsumerId : integer);
begin
  inc(FHead[aConsumerId]);
  if (FHead[aConsumerId] = FBufCount) then
    FHead[aConsumerId] := 0;
end;
procedure TQueuedBuffers.AdvanceTail;
begin
  inc(FTail);
  if (FTail = FBufCount) then
    FTail := 0;
end;
function TQueuedBuffers.qbGetHead(aInx : integer) : PBuffer;
begin
  Result := FBuffers^[FHead[aInx]];
end;
function TQueuedBuffers.qbGetIsNotEmpty(aInx : integer) :
  THandle;
begin
  Result := FIsNotEmpty[aInx];
end;
function TQueuedBuffers.qbGetTail : PBuffer;
begin
  Result := FBuffers^[FTail];
end;
```

➤ *Listing 5: The TQueuedBuffers class for many consumers.*

do much processing per buffer, race ahead and catch up with the producer. We don't have to ensure that all the consumers stay in lockstep, we just have to ensure that they each read the buffers in sequence and don't overtake the producer.

We can model this `TQueueBuffer` object on the one we designed earlier, except that the head pointer is no longer a single entity, since we need one per consumer. An array of them will suffice.

Well, if we have an array of head pointers to create the effect of many queues, it makes sense to have an array of 'queue is not empty' semaphores, one per consumer again. The producer, when it fills another buffer at the tail of the queue, will signal *all* of the 'queue is not empty' semaphores. Apart from these two arrays (and the methods and code needed to service them), the two `TQueueBuffer` classes look much the same.

The producer thread also looks the same. It will wait on the 'queue is not full' semaphore (we'll discuss how this is signalled in a moment), and when it succeeds it will read another buffer from the source stream into the tail buffer and advance the tail pointer. It then signals all of the consumers' 'queue is not empty' semaphores, and goes round the loop again.

The consumer thread has a little more to do. It waits on its 'queue is not empty' semaphore, which will eventually be signalled by the producer. It can then read the data from its own head buffer (recall that there is one per consumer) and then comes the clever bit. The consumers must all read each buffer filled by the producer thread. The consumer that is last to read a particular buffer must signal the producer's 'queue is not full' semaphore. The others can't, since the producer could take that signal as meaning the buffer can now be overwritten, and hence it *must* be the last consumer that does the job.

How can we tell if all the other consumers have read the buffer? Well, the producer, when it fills a buffer at the tail of the queue, will set a field of the buffer to the number of consumers that still have to read the buffer. Obviously, as far as the producer is concerned, that is *all* of the consumers. Each consumer, once it has finished using the buffer at its head, will decrement this count for the buffer. The consumer that decrements this count to zero is obviously the last consumer reading that particular buffer, and hence is the consumer that signals the producer's semaphore.

It Could Happen To You

But (and this is a 'but' that should have struck you forcefully) a particular buffer's count of consumers still to read it is possibly going to be read and updated by several consumer threads at once. This is the classic *race condition* I was talking about last time. Suppose there are two threads, and consumer A reads the count ready to decrement it. It gets the value 2, and decrements it to 1. Before it can write the new value back, consumer B gets control, reads the count, 2, decrements it to 1 and writes it back. Consumer A gets control again and this time manages to write its new value, 1, back. Both threads have finished with the buffer, but the count value is still only 1. The producer is never signalled and we get a deadlock.

No, we must decrement the count in a thread-safe manner. Rather than using a critical section or a mutex, this time we shall use another Win32 primitive, the `InterlockedDecrement` function. This takes the address of a 32-bit value, decrements the value there in a synchronized manner, and returns either zero if the new value was zero, or some other non-zero value if not. It does *not* return the new value, unless that new value happened to be zero, that is. This routine is ideal for our use, it's

```
constructor TProducer.Create(aStream : TStream;
  aBuffers : TQueuedBuffers);
begin
  inherited Create(true);
  FStream := aStream;
  FBuffers := aBuffers;
end;
procedure TProducer.Execute;
var
  Tail : PBuffer;
  i    : integer;
begin
  repeat
    {get the 'queue is not full' semaphore}
    WaitForSingleObject(FBuffers.IsNotFull, INFINITE);
    {read a block from the stream into the head buffer}
    Tail := FBuffers.Tail;
    Tail^.bCount :=
      FStream.Read(Tail^.bBlock, BufferSize);
    Tail^.bToReadCount := FBuffers.ConsumerCount;
    FBuffers.AdvanceTail;    {advance the tail pointer}
    {as we've written a new buffer, signal all the
     'queue is not empty' semaphores}
    for i := 0 to pred(FBuffers.ConsumerCount) do
      ReleaseSemaphore(FBuffers.IsNotEmpty[i], 1, nil);
  until (Tail^.bCount = 0);  {do until stream exhausted}
end;
constructor TConsumer.Create(aStream : TStream; aBuffers :
  TQueuedBuffers; aID : integer);
begin
  inherited Create(true);
  FStream := aStream;
  FBuffers := aBuffers;
  FID := aID;
end;
procedure TConsumer.Execute;
var
  Head      : PBuffer;
  NumToRead : integer;
begin
  {get our 'queue is not empty' semaphore}
  WaitForSingleObject(FBuffers.IsNotEmpty[FID], INFINITE);
  Head := FBuffers.Head[FID];   {get the head buffer}
  {while the head buffer is not empty...}
  while (Head^.bCount <> 0) do begin
    {write a block from the head buffer into the stream}
    FStream.Write(Head^.bBlock, Head^.bCount);
    {we've finished with this buffer, so safely decrement
     count of consumers who have still to read it}
    NumToRead := InterlockedDecrement(Head^.bToReadCount);
    {advance our head pointer}
    FBuffers.AdvanceHead(FID);
    {if we were the last consumer to read this buffer...}
    if (NumToRead = 0) then
      {signal the 'queue is not full' semaphore}
      ReleaseSemaphore(FBuffers.IsNotFull, 1, nil);
    {get our 'queue is not empty' semaphore}
    WaitForSingleObject(FBuffers.IsNotEmpty[FID],
      INFINITE);
    Head := FBuffers.Head[FID];    {get the head buffer}
  end;
end;
procedure AAThreadedMultiCopyStream(aSrcStream : TStream;
  aDestCount : integer; aDestStreams : PaaStreamArray);
var
  i : integer;
  Buffers  : TQueuedBuffers;
  Producer : TProducer;
  Consumer : array [0..pred(aac_MaxConsumers)]
    of TConsumer;
  WaitArray : array [0..aac_MaxConsumers] of THandle;
begin
  Buffers := nil;
  Producer := nil;
  for i := 0 to pred(aac_MaxConsumers) do
    Consumer[i] := nil;
  for i := 0 to aac_MaxConsumers do WaitArray[i] := 0;
  try
    {create the queued buffer object}
    Buffers := TQueuedBuffers.Create(20, aDestCount);
    {create the producer thread, save its handle}
    Producer := TProducer.Create(aSrcStream, Buffers);
    WaitArray[0] := Producer.Handle;
    {create the consumer threads, save their handles}
    for i := 0 to pred(aDestCount) do begin
      Consumer[i] :=
        TConsumer.Create(aDestStreams^[i], Buffers, i);
      WaitArray[i+1] := Consumer[i].Handle;
    end;
    for i := 0 to pred(aDestCount) do
      Consumer[i].Resume;     {start the threads up}
    Producer.Resume;
    {wait for the threads to finish}
    WaitForMultipleObjects(1+aDestCount, @WaitArray, true,
      INFINITE);
  finally
    Producer.Free;
    for i := 0 to pred(aDestCount) do Consumer[i].Free;
    Buffers.Free;
  end;
end;
```

➤ *Listing 6: The producer, the consumer, and the stream multicopy routine.*

extremely fast and it returns a value we can use immediately.

Listing 6 has the final code for the producer thread class in this situation, the consumer thread class, and the routine that ties it all together.

Simply Irresistible

With that we come to the end of an interesting column and the last one I shall do on multithreading until I can get my hands on Kylix. I hope you found it useful and that you can use the ideas and methodology behind the multi-buffered copy routines, if not the routines themselves, in your applications.

Next month, I should have had enough time to judge the entries for the simulated annealing contest and will announce the result.

Julian Bucknall is still hard at work writing the Delphi algorithms book for the new Millennium, and can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.