Here Comes The Flood

by Julian Bucknall

I don't know whether you've ever used TFileStream, but maybe when you did you've thought to yourself that it's, well, slow. I know I have. It's not slow in reading or writing large buffers, but when you are dealing with lots of small buffers (eg bytes, words, booleans, small records and the like) it's downright sluggish. Why? Well, in this article we'll be looking at TFileStream, identifying its positive and negative aspects, and we'll create a new stream descendant that addresses the negative ones.

Family Snapshot

So, TFileStream. It's a descendant of THandleStream, which in turn is a descendant of TStream, the granddaddy of all streams. TFileStream is nothing really special over and above THandleStream: it just defines a new constructor to open a file on disk and the destructor closes the file. As I said, nothing too extraordinary, just enough to be useful when working with files.

THandleStream, on the other hand, is where the nitty-gritty work goes on. The reason that it is supplied is so that you can perform stream operations using the handle to an already open file. It overrides three methods from its ancestor: Read, Write and Seek. If you look in the Classes unit (if you have the source code, that is, if not, trust me) you'll see that Seek performs a seek with the file handle, Read performs a read with the file handle and Write performs a write with the same handle. Pretty simple code. But what's wrong with it? Nothing with its functionality obviously, otherwise Borland would have been deluged with bug reports. However, think about trying to read a byte from the stream. The to the Read method call gets translated to a call to the operating system to read a single byte from the file. You are relying on the operating system to help you out by buffering the file somehow so that the call to read a byte doesn't always translate to a call to move the disk head and read something off the disk platter.

Shaking The Tree

Before moving on, let us sidestep a little and make sure that in this article we are actually going to be making some improvements to TFileStream. Let us set up a test program to profile some code that reads from and writes to a file stream. We'll run it now and get some performance data and then, once we've done, we'll run it again to check that we have actually made an improvement with our work. After all, the only scientific way to show that we have done well is to proffer proof in the shape of some "before and after" statistics. In other words we must be quantitative, not qualitative. Hand-waving arguments are not accepted here.

The test we shall make will be to time 16000 writes of different blocks of 40 bytes and 16000 reads of those written blocks, both forwards and backwards. This test is designed to stress the stream since the blocks are small compared to the disk cluster size. Note that we could use a profiler such as Speed Daemon to perform such tests, however our simple timing will suffice, as we shall see.

The code in Listing 1 creates a new file stream, and then writes 16000 40-byte blocks to the stream (the WriteBlock routine, not shown here, creates a block of 40 copies of StartChar and writes it to the stream). We then seek to the start of the stream, and read 16000 40 byte blocks (ReadCheckBlock, also not shown, reads 40 bytes from the stream and checks that each of the 40 bytes is equal to StartChar). After this, we are positioned at the end of the stream, and so, just for fun, we read the same 16000 blocks but in reverse order. I'm sure you would agree, this would be pretty stressful for a stream.

On my machine at home (Pentium 120 MHz with 32Mb RAM, Windows 95, Maxtor 1.6Gb drive) the above test as a 32-bit program took an average of 8.2 seconds over 5 trials.

Digging In The Dirt

Having seen how quickly (or slowly) the standard file stream works, let us contemplate how to improve it. The answer seems obvious: somehow buffer up a bunch of reads of small blocks as a single read of many bytes and then dole out the smaller blocks as and

➤ *Listing 1*

```
S := TFileStream.Create('Test', fmCreate);
try
  StartTime := GetTickCount;
  writeln('writing blocks');
  StartChar := 'A';
  for i := 0 to 15999 do
    WriteBlock(S);
  writeln('seek to start');
  S.Seek(0, soFromBeginning);
  writeln('readin/checking blocks');
  StartChar := 'A';
  for i := 0 to 15999 do
    ReadCheckBlock(S);
  writeln('readin/checking blocks in reverse order');
  S.Seek(0, soFromEnd);
  for i := 0 to 15999 do begin
    S.Seek(-BytesInBlock, soFromCurrent);
    dec(StartChar);
    ReadCheckBlock(S);
    S.Seek(-BytesInBlock, soFromCurrent);
    dec(StartChar);
  end;
  writeln('Time taken: ', GetTickCount-StartTime);
finally
  S.Destroy;
end;
```

when required. Similarly for writes. In other words, we need to buffer our file data. As it happens, Delphi already has a buffered stream. Luckily for me (otherwise you would not be reading this article) this buffered stream is embedded so deeply into the internal TReader and TWriter classes that to extract it would take some work. For information about TReader and TWriter, you should look to your Delphi help file or browse the Classes unit. We shall not discuss them further here.

Now, buffering reads and writes sounds simple enough, but we cannot be certain how our buffered file stream will be used. It may be that we have 1000 writes followed by 1000 reads, but then again we may have reads, writes and seeks in some 'random' order. We cannot count on an orderly universe for our new stream, so the buffering we envisage must be clever enough to encompass reads and seeks and writes in any order.

The design I propose is a class that has a memory block that buffers data from the file. If we want 40 bytes, for example, the class will read a bigger chunk from the file. The class will pass back the 40 bytes requested and store the rest of the block in case it's needed by a further read. It makes sense to make the buffer memory block some multiple or fraction of a disk cluster, say 4Kb or 8Kb or whatever. This will help maximize throughput to the actual disk. Going even further than this, it also makes sense for the class to read from (or write to) the file as if the file were a file of records, each record being a block in size. This would mean that no block that we read straddles a disk cluster boundary and thus improving our throughput again (remember that disk fragmentation means that clusters in a file may be separated by a great acreage of disk platter).

Summing up, we shall treat the file as a file of equal sized blocks or pages. Each page will be sized based on the disk cluster size and in practice this means the pages are 4Kb or 8Kb in size. Various tests I've run have shown that increasing the page size from 4Kb or 8Kb to 16Kb or 32Kb does not appreciably improve the performance of the buffered handle stream. You can think of the buffer as a 'window' on the file. We'll be moving the 'window' to and fro over the file depending on our position in the stream.

Internally, we need to remember various things. Has the buffer been written to? In other words, do we need to write the buffer to the file at some point? This state is determined by an internal variable which is traditionally known as the dirty flag. When we write data to the buffer, we set the flag true. When we write the buffered data to disk, we set it false. Which page of the file does the buffer represent? This is simple, we must store the starting offset of the page in the file. Next, what about the stream position? We must be able to calculate this on demand. It will consist of the starting offset of the page (which we're already storing) and the position within the buffer, so the latter is another value we shall have to monitor. Finally, we need to know how much of the buffer is being used. Imagine creating a new stream and writing 40 bytes to it. Obviously, if we assume a 4Kb buffer, we have 40 bytes of valid data and 4056 bytes of spare space waiting to be filled. If we close the stream at this point, we need to write only 40 bytes to disk, not the whole 4Kb buffer. This determines another variable: the number of valid bytes in the buffer.

That takes care of some of the internal aspects of the buffered stream class, so let us consider the external aspect. This is pretty easy: we're creating a descendant of a TStream, so we must override the methods that have been declared virtual. These are:

```
function Read(var Buffer;
  Count : Longint) : Longint;
  override;
 {read from stream into buffer}

function Write(const Buffer;
  Count : Longint) : Longint;
  override;
 {write to stream from buffer}

function Seek(Offset : Longint;
  Origin : Word) : Longint;
  override;
 {seek to a particular point
  in the stream}
```

In Delphi 3, there is yet another virtual method that needs overriding:

```
procedure SetSize(NewSize :
  Longint); override;
  {set the stream size}
```

With this we come to the end of our design and it is time to start coding. We shall descend our TbhsBufferedHandleStream class from TStream instead of THandleStream because we shall effectively be replacing all of THandleStream's methods.

And Through The Wire

Simple things first. The Create constructor is passed a handle (obviously!) and also the size of the buffer required. The method does some calculation on the passed buffer size to round it up to the nearest multiple of 1Kb with an absolute maximum of 32Kb. A buffer is then allocated from the heap, some internal variables are set and then the code finds out the file size: this will be the starting size of the stream. The Destroy destructor frees the buffer, after making sure that if the buffer is dirty it is written to disk first.

The next simplest method to discuss is the Seek method. This method is supposed to position the stream at a given point. The measurement is taken either from the beginning of the stream, from the end or from the current position (this is the Origin parameter). The method works out the new position of the stream based on the passed parameters and raises an exception if it does not fall within the stream's boundaries. All fine so far, now it gets interesting. From this new value we can work out the page of the file this new position appears in. If this is the same as the page that's currently in the buffer, all well and good. If it is not then we have to mark the page as discarded: this is simple enough, we simply set the count of valid bytes as zero. This will trigger the page to be read later on. But, wait a minute, what happens if we've just written something in this page? Before marking the page as discarded or "not present" we need to write it out to the file itself.

Now, like it or not, we need to discuss the `Read` and `Write` methods. These are easily the most complex portions of our new stream class mainly because of all the housekeeping that needs to go on. But first a point about our implementation of them. Both methods take a buffer and a count of bytes to read into or write from that buffer, and they return a count of bytes actually read or written. Our `Read` method will read up to the end of the stream and will return a correct number of bytes read. If, however, a problem occurs when reading from the file, an exception will be raised rather than returning a lesser number of bytes read. When writing, the returned number of bytes written will always be equal to the amount requested. If an error occurs whilst writing (eg the disk becomes full) an exception will be raised rather than returning a lesser number of bytes written. Also in 16-bit, I am artificially limiting the number of bytes that can be read or written to less than 64Kb: I didn't want to be bothered with selector arithmetic for this article.

Having made all that clear, let's move onto the `Read` method (see Listing 2). It starts innocuously enough by calculating the number of bytes it will actually read, that being the lesser of that requested or the remainder in the stream. Next it makes sure that the buffer has some data in it; if not, the method calls the `bhsReadBuffer` method to read a page from the file into the buffer. Now the going gets a little tougher. The requested number of bytes to read can either be satisfied completely from the current page in the buffer (remember that we have a variable which holds the number of valid bytes in the buffer), or it will span two or more pages. If the former, then things go pretty easily: move the required number of bytes from the buffer (making sure we copy from the current position in the buffer) to the caller's memory block, advance our position in the buffer and we're done. If the latter, then we need to replenish our buffer at some stage. Copy the rest of this buffer's worth of data over to the caller's memory block. Make a note of where we are in this block. Now advance our buffer's starting offset and read the next page in from the file. But wait! What if the buffer is dirty? In other words we have written some new data to this page, but it hasn't yet been written to disk. Well, no surprise here, we write the current buffer to disk before reading the next. We continue this process until all the requested bytes have been read from the stream. It may mean just this one extra buffer read or, if the amount of data requested is large enough, it may mean several.

Shock The Monkey

Now the `Write` method. Having discussed the `Read` method we can see that the `Write` method (Listing 3) works in roughly the same way. But, that is just where I went wrong, and where all of my bugs were found when I was writing and testing the code. Oh, the perils of copy and paste! Anyway, we start off by calculating the number of bytes we shall write. As I said before, we assume we shall write all the bytes requested; if we can't, for example if the disk is full, we'll raise an exception. Next, we make sure that the buffer has some data in it. Here came the first bug: I was using the count of valid bytes in the buffer as an indication that the buffer needed to be replenished, if this value was zero, then we need to read the page into the buffer. However, it is entirely possible that the stream is an exact number

➤ *Listing 2*

```
function TbhsBufferedHandleStream.Read(var Buffer;
  Count : Longint) : Longint;
var
  BufAsBytes  : TByteArray absolute Buffer;
  BufInx      : Longint;
  BytesToGo   : Longint;
  BytesToRead : integer;
begin
 {$IFDEF Windows}
  {in Delphi 1 we do not support reads over 65535 bytes}
  if (Count > $FFFF) then
    RaiseException('TbhsBufferedHandleStream.Read: '+
      'requested too many bytes');
 {$ENDIF}
  {calculate actual number of bytes we can read - depends on
   current position and size of stream as well as number
   of bytes requested}
  BytesToGo := Count;
  if (bhsSize < (bhsPageStart + bhsPosInPage + Count)) then
    BytesToGo := bhsSize - (bhsPageStart + bhsPosInPage);
  if (BytesToGo <= 0) then begin
    Result := 0;
    Exit;
  end;
  {remember to return the result of our calculation}
  Result := BytesToGo;
  {initialise the byte index for the caller's buffer}
  BufInx := 0;
  {is there anything in the buffer? if not, go read
   something from the file on disk}
  if (bhsByteCount = 0) then
    bhsReadBuffer;
  {calculate number of bytes we can read prior to the loop}
  BytesToRead := bhsByteCount - bhsPosInPage;
  if (BytesToRead > BytesToGo) then
    BytesToRead := BytesToGo;
  {copy from the stream buffer to the caller's buffer}
  Move(bhsPage^[bhsPosInPage], BufAsBytes[BufInx],
    BytesToRead);
  {calculate the number of bytes still to read}
  dec(BytesToGo, BytesToRead);
  {while we have bytes to read, read them}
  while (BytesToGo > 0) do begin
    {advance the byte index for the caller's buffer}
    inc(BufInx, BytesToRead);
    {as we've exhausted this buffer-full, advance to next,
     check to see if we need to write the buffer out first}
    if bhsDirty then begin
      bhsWriteBuffer;
      bhsDirty := false;
    end;
    inc(bhsPageStart, bhsPageSize);
    bhsPosInPage := 0;
    bhsReadBuffer;
    {calculate number of bytes we can read in this cycle}
    BytesToRead := bhsByteCount;
    if (BytesToRead > BytesToGo) then
      BytesToRead := BytesToGo;
    {copy from the stream buffer to the caller's buffer}
    Move(bhsPage^, BufAsBytes[BufInx], BytesToRead);
    {calculate the number of bytes still to read}
    dec(BytesToGo, BytesToRead);
  end;
  {remember our new position}
  inc(bhsPosInPage, BytesToRead);
  if (bhsPosInPage = bhsPageSize) then begin
    inc(bhsPageStart, bhsPageSize);
    bhsPosInPage := 0;
    bhsByteCount := 0;
  end;
end;
```

of pages long. In other words, it is entirely valid at this point for there to be no more data in the stream. The better test is to not only check that the number of bytes is zero but also that the stream size is larger than the current page offset.

Next, we calculate the number of bytes in the buffer that we can fill. Again another bug appeared whilst I was coding. In the `Read` routine the number of remaining bytes is the difference between where we are in the buffer and the number of valid bytes in the buffer. For writing, it is the difference between where we are in the buffer and the end of the buffer. A subtle difference but important.

On we go: again we have a choice, there might be enough room in the buffer to accommodate this particular write, or we may have to span the write over two or more buffers full. If the former, we copy over the data from the caller's buffer, set the dirty flag (ie the buffer now contains data that needs to be written to the file) and move onto the last part of the routine. If the latter, we fill this buffer from the caller's and then set the dirty flag. We write out this buffer full, advance our page offset, see whether we can read any more from the file, and then copy more data from the caller's buffer into ours. We continue this process until we have completed copying data over and writing out full buffers to the file. Eventually we arrive at the last part of the routine where we need to make sure that our various state variables are corrected. First we remember our position in the buffer, then we set the count of valid bytes in the buffer. We may have been appending data to the end of the stream, so we then make sure that our stream size variable is correct. Finally we make sure that we are not positioned at the end of the buffer, by writing it out if we are. And that's it.

The `SetSize` method is simplicity itself. It truncates the file at the size indicated and makes sure that the stream position is still within the file's boundaries. I've made sure that `SetSize` is available for Delphi 1 and 2 as well as Delphi 3.

In the `Buffstrm` unit that is provided on the disk there is one more class: the `TbfsBufferedFileStream` class. This is a descendant of the `TbhsBufferedHandleStream` class that takes care of opening and closing a file for you and creating the handle used by the ancestor. It is a pretty simple class: the `Create` constructor opens the file or creates it and the `Destroy` destructor closes it. Much like the `TFileStream` class builds on the `THandleStream` class, as a matter of fact.

Big Time

And so, did all this design and coding heartache make a difference? I reran the same test program as before but using a `TbfsBufferedFileStream` instead of a `TFileStream`. The 32-bit test took an average (over 5 trials again) of 0.43 seconds. That's 19 times faster. So, bam, we hit the big time.

Julian Bucknall works for TurboPower Software in real life. In his not-so-real life, his so-called spare time, he acts a lot in local productions. He can be reached by email at julianb@turbopower.com or on CompuServe at 100116,1572. The code that accompanies this article is freeware and can be used as is in your own applications.

➤ *Listing 3*

```
function TbhsBufferedHandleStream.Write(const Buffer;
  Count : Longint) : Longint;
var
  BufAsBytes   : TByteArray absolute Buffer;
  BufInx       : Longint;
  BytesToGo    : Longint;
  BytesToWrite: integer;
begin
 {$IFDEF Windows}
  {in Delphi 1 we do not support writes greater than
   65535 bytes}
  if (Count > $FFFF) then
    RaiseException('TbhsBufferedHandleStream.Write: '+
      'requested too many bytes');
 {$ENDIF}
  {when we write to this stream always assume that can write
   requested number of bytes: if we can't (eg disk full)
   we'll get an exception somewhere eventually}
  BytesToGo := Count;
  {remember to return the result of our calculation}
  Result := BytesToGo;
  {initialise the byte index for the caller's buffer}
  BufInx := 0;
  {is there anything in the buffer? if not, try read a block
   from file on disk - we might be overwriting existing data
   rather than appending data to the end of the stream}
  if (bhsByteCount = 0) and (bhsSize > bhsPageStart) then
    bhsReadBuffer;
  {calculate number of bytes we can write prior to the loop}
  BytesToWrite := bhsPageSize - bhsPosInPage;
  if (BytesToWrite > BytesToGo) then
    BytesToWrite := BytesToGo;
  {copy from the caller's buffer to the stream buffer}
  Move(BufAsBytes[BufInx], bhsPage^[bhsPosInPage],
    BytesToWrite);
  {mark stream buffer as requiring a save to disk, note this
   will suffice for rest of the routine as well: no inner
   routine will turn off the dirty flag}
  bhsDirty := true;
  {calculate the number of bytes still to write}
  dec(BytesToGo, BytesToWrite);
  {while we have bytes to write, write them}
  while (BytesToGo > 0) do begin
    {advance the byte index for the caller's buffer}
    inc(BufInx, BytesToWrite);
    {as we've filled this buffer, write it out to disk and
     advance to the next buffer, reading it if required}
    bhsByteCount := bhsPageSize;
    bhsWriteBuffer;
    inc(bhsPageStart, bhsPageSize);
    bhsPosInPage := 0;
    bhsByteCount := 0;
    if (bhsSize > bhsPageStart) then
      bhsReadBuffer;
    {calculate number of bytes we can write in this cycle}
    BytesToWrite := bhsPageSize;
    if (BytesToWrite > BytesToGo) then
      BytesToWrite := BytesToGo;
    {copy from the caller's buffer to the stream buffer}
    Move(BufAsBytes[BufInx], bhsPage^, BytesToWrite);
    {calculate the number of bytes still to write}
    dec(BytesToGo, BytesToWrite);
  end;
  {remember our new position}
  inc(bhsPosInPage, BytesToWrite);
  {make sure the count of valid bytes is correct}
  if (bhsByteCount < bhsPosInPage) then
    bhsByteCount := bhsPosInPage;
  {make sure the stream size is correct}
  if (bhsSize < (bhsPageStart + bhsByteCount)) then
    bhsSize := bhsPageStart + bhsByteCount;
  {if we're at the end of the buffer, write it out and
   advance to the start of the next page}
  if (bhsPosInPage = bhsPageSize) then begin
    bhsWriteBuffer;
    bhsDirty := false;
    inc(bhsPageStart, bhsPageSize);
    bhsPosInPage := 0;
    bhsByteCount := 0;
  end;
end;
```