Shared Pools

And I ain't talking about summer and swimsuits there...

by Primoz Gabrijelcic

Welcome to the fourth instalment of this occasional series dealing with interprocess communication, data sharing, Windows NT security and other obscure topics.

For my new readers, and others who want to refresh their memory, a quick recap. In previous articles I've covered some basic synchronisation primitives (Issue 86), presented an intelligent shared memory class (Issue 88), and covered some NT security issues (Issue 91).

This article builds mostly on the second part (shared memory). Actually, it was meant to be the third part in the series, but then things went wrong and the code was not working and, well, if you have read the article in Issue 91 you already know all that. But now the bugs are removed and the shared pool is working as good as it should (which may not be as good as it *could*, but that is for you to fix, if you accept the task).

As powerful as the shared memory implementation from Issue 88 is, it still doesn't solve all the problems Windows puts in our path. Sometimes using shared memory is still a non-trivial thing. Sometimes we run into problems because of a Windows feature, and most of the time this is a very good feature, that a file mapping (an essential part of my shared memory implementation) cannot exist on its own. A file mapping, like mutexes, events, and other Windows primitives, must have an owner. If all processes associated with a given file mapping die, the file mapping will be destroyed. Because in my shared memory implementation this file mapping is backed with a page file, its contents won't be preserved on an accessible part of the disk. In other words, the shared memory contents will be lost.

As I've already said, this is usually 'a good thing'. When a process dies unexpectedly, we expect the Good Operating System to clean up behind it. That includes destroying unused file mappings.

So why does this present a problem? Let's say we have a client-server application (running on one computer, of course, or shared memory wouldn't be of much significance). Clients are generating data and the server is processing it. The clients are delivering data in large chunks. In that scenario, shared memory can be used to transfer data from the client to the server. However, there are difficulties.

Due to the requirement we mentioned before, at all times somebody must own the shared memory buffers carrying the data. From the client's viewpoint, that means that it must keep the shared memory open until the server receives it. If the server only sends this confirmation when the shared memory buffer is processed (and that may take a while as there may be other buffers from other clients waiting), things may get complicated, for the server and also for the clients.

To simplify the programming, I have encapsulated the required client-server protocols into the shared memory pool class (or, I should say, classes). You can therefore concentrate on the server and client sides and the shared memory pool will bring data safely from the clients to the server.

Architecture

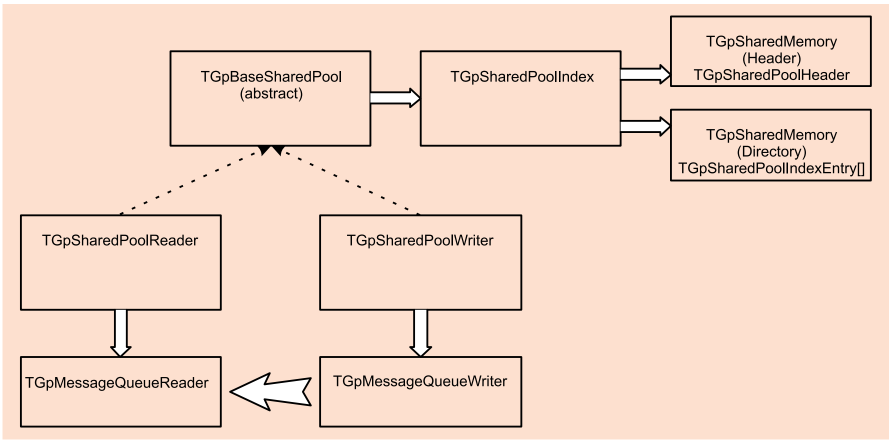
Having said all that, it is not surprising to know that the shared memory pool is also designed in a client-server manner.

One server object (called a `Reader` because it processes the data) manages (allocates, frees) memory buffers in the pool and owns them at all times. Clients (`Writers`) can only acquire a pre-allocated buffer from this server, write the data into the buffer and return it for processing.

Both `Reader` and `Writer` derive from the same base class, `TGpBaseSharedPool` (see Figure 1). This class is quite large as it implements most of the `Writer` functionality. `Reader` is implemented in the `TGpSharedPoolReader` class and `Writer` in the `TGpSharedPoolWriter` class. Both end classes, `Reader` and `Writer`, use a message queue object (`TGpMessageQueueReader/Writer`) to pass messages safely over the OS-provided boundaries (see Issue 91 for more detail).

Pool administration, global values, the list of allocated buffers and so on is separated into a special class `TGpSharedPoolIndex`,

➤ *Figure 1: Shared pool architecture.*


TGpBaseSharedPool (abstract)
TGpSharedPoolIndex
TGpSharedMemory (Header) TGpSharedPoolHeader
TGpSharedMemory (Directory) TGpSharedPoolIndexEntry[]
TGpSharedPoolReader
TGpSharedPoolWriter
TGpMessageQueueReader
TGpMessageQueueWriter

which owns two shared memory objects, one holding the pool header (global values) and another the pool directory (the list of allocated buffers).

For better adaptability, the pool is resizable. Reader starts with some buffers pre-allocated and then adjusts the number of buffers according to usage dynamics and programmer-specified boundary values. When the number of free buffers drops below some limit, Reader allocates more buffers (up to some preset maximum number). When the number of unused buffers grows, Reader will free them. Buffers may also be allocated on request, when a Writer needs a buffer and there is none available, it will ask the Reader to kindly provide another buffer if possible. In that case the new buffer will be allocated immediately.

The header is internally represented by the TGpSharedPoolHeader structure (Listing 1), which is stored in the shared memory object. The code accesses the header shared memory when required, aliases a PGpSharedPoolHeader pointer to the shared memory data and works with the members directly.

As you can see from the code, the header is divided into three parts: static members, dynamic members and list headers. I'll cover the list headers later, and dynamic members are trivial, but the static part deserves to be described in more detail.

The first member (sphSignature) contains a special number which the TGpSharedPoolIndex uses as a safety check, to prevent it from accidentally connecting to some other shared memory owned by an unrelated application that just happens to have the 'right' name. The second member is a header version, currently always 1. If the header is extended with additional members in some future release, this version will be incremented.

The next seven members, from sphInitialBufferSize to sphResizeThreshold, contain a copy of the programmer initialisation parameters provided by the programmer, passed to the Reader's Initialize method. We'll cover those later.

The last three static members are probably the most interesting. The sphOwnersToken holds a Reader's token (unique name, see Issue 86), which is used to generate shared memory buffer names and to detect whether the server has been restarted. Next, sphIndexEntrySize holds the size of one entry in the pool directory and sphNumEntries holds the total number of entries in the same directory.

The reason for the sphIndexEntrySize may not be obvious at first glance. After all, the code knows how big the entry is. That is only partially true: the Reader, which created the header, knows how big this entry is. The Writer, on the other hand, may run the older version of the pool software when the index entry was smaller. Thus, storing the entry size in the header allows all clients to reliably access any entry in the directory.

```
TGpSharedPoolHeader = packed record
//static
  sphSignature        : int64;       // "magic" signature
  sphVersion          : cardinal;    // version; currently always 1
  sphInitialBufferSize: cardinal;    // initial size of shared memory buffers
  sphMaxBufferSize    : cardinal;    // maximum size of shared memory buffers
  sphMinBuffers       : cardinal;    // minimum number of memory buffers
  sphMaxBuffers       : cardinal;    // maximum number of memory buffers
  sphSweepTimeoutSec  : cardinal;    // number of seconds buffer must be left
untouched before it is disposed
  sphResizeIncrement  : cardinal;    // number of buffers to be allocated when
resizing
  sphResizeThreshold  : cardinal;    // number of free buffers when resize
should be triggered
  sphOwnersToken      : array [0..263] of char; // name of the owner's token;
used to generate shared memory buffer names
  sphIndexEntrySize   : cardinal;    // size of the index entry
  sphNumEntries       : cardinal;    // number of entries in the directory
//dynamic
  sphNumBuffers       : cardinal;    // number of memory buffers
  sphFreeBuffers      : cardinal;    // number of free memory buffers
//list headers
  sphFreeList         : cardinal;    // index of the last used free entry
  sphQueuedList       : cardinal;    // index of the first queue entry
  sphDisposedList     : cardinal;    // index of the first disposed entry
end; { TGpSharedPoolHeader }
```

➤ *Listing 1: Shared pool header.*

➤ *Listing 2: Directory entry.*

```
TGpSharedPoolIndexEntry = packed record
  spieStatus    : integer;         // status (free, allocated, queued, disposed)
  spieReleasedAt: TDateTime;       // time then buffer was released
//links
  spiePrevious  : cardinal;        // link to previous entry with the same status
  spieNext      : cardinal;        // link to next entry with the same status
//internals
  spieHandle    : TGpSharedMemory; // pointer to shared memory buffer on *reader*
end; { TGpSharedPoolIndexEntry }
```

The directory is a shared memory object containing array of entries of type TGpSharedPoolIndexEntry (Listing 2). There are sphNumEntries entries in the directory. That also means that at most sphNumEntries buffers can participate in the pool.

The spieStatus member holds the status of this entry (and the associated memory buffer). It can either be free (the memory buffer is allocated but doesn't hold any data), allocated (the memory buffer is acquired by a Writer), queued (the memory buffer was submitted to the Reader for processing), or disposed (the memory buffer is not allocated).

The spieReleasedAt keeps the time when the buffer was released. This is used when the 'sweeper' (described later) shrinks the pool.

The next two members, spiePrevious and spieNext, link this entry to the previous and next entries with the same status. The first and last entries in this doubly linked list have its spiePrevious (spieNext) member set to -1 to indicate the end of the list.

Lastly, the spieHandle member holds the handle of the shared memory buffer (when the buffer is allocated). This handle has meaning only inside the Reader's process.

Now the meaning of 'list header' header members becomes clear: they hold the index of the first entry in the free/queued/disposed lists. There is no sphAllocateList member because 'allocated' entries are not connected into a list. When an entry is allocated, it is removed from the lists and is in complete control of the Writer which owns it.

Creation

Although a Writer can be created before the Reader it cannot do much useful work alone, because it can't acquire a memory buffer. The reason for that is obvious: the Reader creates and owns memory buffers, if there is no Reader, there are no buffers. The Writer merely creates a TGpMessageQueueWriter (which it will later use to signal its needs to the Reader) and the shared index object (but it will not initialise the header).

The Reader doesn't do much more during the object creation, it will create some communication objects and the shared index object (again, uninitialised). Only when the code calls the Initialize method (Listing 3) does the Reader spring into life.

As you can see, there are lots of configuration parameters, but only four must be provided. The first two, initialBufferSize and maxBufferSize, regulate the initial and maximum size for the managed memory buffers. (Shared memory objects are either resizable, in which case initial and maximum size must be larger then zero, or fixed, when the initial size is positive and the maximum size is zero. For more information see Issue 88.)

The next two parameters regulate the initial and maximum numbers of memory buffers in the pool. I have no idea what problem you're solving and what numbers to put in there. I would only suggest that you don't set the maximum number too low, as unused memory buffers don't take much space, and that you set the initial number of buffers to a non-zero value (mostly because this number is also used for various resize defaults).

The other four parameters are optional. The first two regulate the pool growth. Additional buffers are allocated if the number of free buffers falls below the resizeThreshold value. If not set, this value is calculated automatically with a quite convoluted formula (if you just have to know, see TGpSharedPoolIndex.Initialize). The next parameter (resizeIncrement) specifies how many additional buffers are allocated in such a case (of course, the total number of buffers will never grow above the maxBufferSize). This value defaults to startNumBuffers. The third parameter (minNumBuffers) represents the minimal allowed number of buffers in the pool. Leaving it at the default means that the number of allocated buffers will never fall below startNumBuffers.

The last parameter, sweepTimeoutSec, is an interval between consecutive passes of the 'sweeper', the part of the code that checks the pool and frees buffers that were not used for a long time (the default is 15 seconds).

Besides that, the initialization code also creates a mutex (to tell the world the server is alive) and a TGpMessageQueueReader to receive the Writer's commands and data.

➤ *Listing 3: Signature of the Reader's initialization method.*

```
function TGpSharedMemoryReader.Initialize(initialBufferSize, maxBufferSize,
  startNumBuffers, maxNumBuffers: cardinal; resizeIncrement: cardinal =
  CDefResizeIncrement; resizeThreshold: cardinal = CDefResizeThreshold;
  minNumBuffers: cardinal = CDefMinNumBuffers; sweepTimeoutSec: cardinal =
  CDefSweepTimeoutSec): boolean;
```

Acquiring A Buffer

To acquire a buffer, Writer calls the AcquireBuffer method, which takes one parameter, timeout. If the request cannot be completed inside this limit (specified in milliseconds, but you can also use the constant INFINITE), the function will return nil. There are other cases where the AcquireBuffer can fail, so you should always check the error code (it is stored in the LastError property). If the function succeeds, it will return the TGpSharedMemory object into which you can put the data (see Issue 88 for more details).

AcquireBuffer actually only calls the InternalAcquireBuffer (Listing 4) and tells it to acquire the index object. The first thing the internal version does is exactly that, it will acquire the index. This may fail if the index is not initialised (because the Reader has not been started yet) or if the index is initialised but the Reader is not alive (the

➤ *Listing 4: Acquiring a buffer.*

```
function TGpBaseSharedPool.InternalAcquireBuffer(timeout:
  DWORD; doAcquireIndex: boolean): TGpSharedMemory;
var
  start: int64;
begin
  Result := nil;
  SetError(speOK);
  start := GetTickCount;
  repeat
    if doAcquireIndex and (not AcquireIndex) then
      SetError(speNoReader)
    else begin
      try
        if IsReader and (Index.FreeBuffers = 0) then
          UnprotectedTryToResize;
        if Index.FreeBuffers > 0 then begin
          Result := Index.GetFreeBuffer;
          if Index.ShouldResize then
            if IsReader then
              UnprotectedTryToResize
            else
              MessageQueue.PostMessage(
                CGpSharedPoolMessageQueueWriteTimeout,
                WM_PLEASE_RESIZE, 0, 0);
        end else if Index.CanResize then begin
          assert(not IsReader,'GpSharedMemory/
            TGpBaseSharedPool.AcquireMemory: IsReader and
            Index.CanResize');
          MessageQueue.PostMessage(
            CGpSharedPoolMessageQueueWriteTimeout,
            WM_PLEASE_RESIZE, 0, 0);
        end;
      finally
        if doAcquireIndex then
          ReleaseIndex;
      end;
    end; //else not AcquireIndex
    if assigned(Result) or Elapsed(start, timeout) then
      break; //repeat
    Sleep(0);
  until false;
  if (not assigned(Result)) and (LastError = speOK) then
    SetError(spePoolFull);
  if assigned(Result) then
    bspAcquiredList.Add(Result);
end; { TGpBaseSharedPool.InternalAcquireBuffer }
```

code checks if the mutex created in Initialize is alive).

Next we can afford some optimisation. If the code was called from the Reader (bear with me just a second, I know that the Writer is the one that should be calling the method) and the number of free buffers is zero, we can try to allocate more buffers immediately.

If the number of free buffers is now positive, we can allocate the buffer by calling Index.GetFreeBuffer. If after that the number of free buffers falls below the resizeThreshold, we can do one of two things. If running on the Reader, the code will simply call the resizing method, but if running on the Writer, this is not possible as the resizing code must execute on the Reader. The Writer can only post a message WM_PLEASE_RESIZE to the Reader. To overcome the various service-related problems (described in Issue 91), a message is sent using my TGpMessageQueueWriter object.

However, if the number of free buffers is zero, we cannot do much better than post WM_PLEASE_RESIZE to the Reader. A small peek in the index checks whether all the buffers are already allocated. In that case, there is no way the resize could succeed and the message is not posted at all.

At the end, the index is released and the code loops to the beginning if the buffer was not acquired and the timeout has not yet been exceeded. Yes, the code is polling the pool here. Maybe the problem could be solved without that, using some global event that gets triggered whenever the number of free buffers changes, but I have a feeling that the code is too complicated already. After all, in a well-designed system you should almost never run out of buffers.

At the end, the newly allocated buffer (if any) is added to the internal list. When the object is destroyed, all the buffers in this list are returned to the pool automatically.

So what is that with the IsReader checks? Basically, every Reader also implements a complete Writer interface. The Reader can use this to send some partially processed buffers back to self for further

➤ *Listing 5: Retrieving a free buffer from the index.*

```
function TGpSharedPoolIndex.GetFreeBuffer: TGpSharedMemory;
var
  freeBuffer: cardinal;
  header    : PGpSharedPoolHeader;
begin
  if FreeBuffers = 0 then
    raise Exception.Create(
      'GpSharedMemory/TGpSharedPoolIndex.GetFreeBuffer: no free buffers');
  header := SafeGetHeader('GetFreeBuffer');
  freeBuffer := header^.sphFreeList;
  Unlink(freeBuffer,header^.sphFreeList);
  Dec(header^.sphFreeBuffers);
  Entry[freeBuffer].spieStatus := Ord(iesAllocated);
  Result := TGpSharedMemory.Create(GetEntryName(freeBuffer),
    BufferInitialSize, BufferMaxSize, false);
  Result.AcquireMemory(true, 0);
end; { TGpSharedPoolIndex.GetFreeBuffer }
```

processing. Or, you can use this to implement a Reader/Writer interface with only one object. Or... I'm sure you'll find another use for it.

At the class level this is implemented by putting all the Writer functionality into the base class TGpBaseSharedPool from which Reader class TGpSharedPoolReader (with added reading functionality) and Writer class TGpSharedPoolWriter (with almost no addition) are derived.

Another interesting method, called from the InternalAcquireBuffer, is TGpSharedPoolIndex.GetFreeBuffer (Listing 5). It is pretty straightforward but serves as a good example of how the index is manipulated. First, the code checks if there is a free buffer to allocate (to catch programming errors). Next it retrieves the pointer to the header memory and an index of the first directory entry in the free list (header^.sphFreeList) and removes it from this list (Unlink). The number of free buffers is then decremented, the status of this buffer is changed to allocated and a new shared memory object is created. The name of this buffer is generated (via GetEntryName, Listing 6) from the Reader's token (I am not ready to discuss the reason for that token), plus the intermediate /Pool/ and directory index of this entry. The last part is used to locate this shared memory in the pool given only the name (GetEntryIndex, also in Listing 6).

Please Process That

When the Writer is done with the buffer returned from the AcquireBuffer, it can send it back to the Reader with a call to the SendBuffer method (see Listing 7). It takes one parameter (TGpSharedMemory object) and returns True or False. If there's an error, additional information is stored in the LastError property.

Following the AcquireBuffer pattern, SendBuffer first acquires the index. It then calls the TGpSharedPoolIndex method PrepareToSend (Listing 8), which converts the shared memory object's name back to the directory index, retrieves that entry, checks if it really is allocated (to catch programming errors), links it into the 'queued' list, and frees the shared memory object.

That last action may seem too hasty, as the buffer was not received by the Reader yet, but we must never forget the larger picture. First, the Reader creates a shared memory object and underlying Windows file mapping. Then a Writer acquires this shared memory by creating *another* shared memory object that uses the *same* file mapping as the first object. That is why the memory handled by this object is 'shared': both sides access the same file mapping and therefore the same memory.

When the Writer destroys this shared memory object, the Reader's shared memory is still alive. Therefore, the underlying file mapping is still active and our data is safe.

Let's return back to the SendBuffer. After calling PrepareToSend, it removes the shared memory object from the internal list of allocated objects. Only then is the PrepareToSend error code checked. If there was no error, SendBuffer simply posts WM_DATA_SENT to the Reader (even when called from the Reader itself). The Reader will react to this message by scanning the directory for all waiting buffers (neatly collected in the 'queued' list) and processing them.

PrepareToSend, however, can return an error. Specifically, if this error is speNotOwner (indicating that we are trying to send a buffer that is not owned by the Reader), then the SendBuffer will allocate new buffer and copy data from the existing buffer into it, remove and destroy the buffer you are trying to send, and at the end pass the newly created buffer to the PrepareToSend method.

➤ *Listing 6: Converting index into name and back.*

```
function TGpSharedPoolIndex.GetEntryName(idx: cardinal): string;
begin
  Result := SafeGetHeader('GetEntryName')^.sphOwnersToken+'/Pool/';
  if idx <> INVALID_HANDLE_VALUE then
    Result := Result + IntToStr(idx);
end; { TGpSharedPoolIndex.GetEntryName }
function TGpSharedPoolIndex.GetEntryIndex(name: string): cardinal;
var
  namePrefix: string;
begin
  Result := INVALID_HANDLE_VALUE;
  namePrefix := GetEntryName(INVALID_HANDLE_VALUE);
  if StrLIComp(PChar(name),PChar(namePrefix),Length(namePrefix)) = 0 then begin
    Delete(name,1,Length(namePrefix));
    Result := cardinal(StrToIntDef(name,integer(INVALID_HANDLE_VALUE)));
  end;
end; { TGpSharedPoolIndex.GetEntryIndex }
```

➤ *Listing 7: Sending the buffer back.*

```
function TGpBaseSharedPool.SendBuffer(var shm:
  TGpSharedMemory): boolean;
var
  tempBuf: TGpSharedMemory;
begin
  if not AcquireIndex then
    Result := SetError(speNoReader)
  else begin
    try
      tempBuf := shm;
      Result := SetError(Index.PrepareToSend(shm));
      if Result then
        bspAcquiredList.Remove(tempBuf)
      else if LastError = speNotOwner then begin
        tempBuf := InternalAcquireCopy(shm,
          CSharedPoolForeignSendTimeoutSec*1000, false);
        Result := assigned(tempBuf);
        if Result then begin
          bspAcquiredList.Remove(shm);
          FreeAndNil(shm);
          shm := tempBuf;
          Result := SetError(Index.PrepareToSend(shm));
          if Result then
            bspAcquiredList.Remove(tempBuf);
        end;
      end; //if Result = speNotOwner
      if Result then
        MessageQueue.PostMessage(
          CGpSharedPoolMessageQueueWriteTimeout,
          WM_DATA_SENT, 0, 0);
    finally ReleaseIndex; end;
  end;
end; { TGpBaseSharedPool.SendBuffer }
```

```
function TGpSharedPoolIndex.PrepareToSend(
  var buffer: TGpSharedMemory): TGpSharedPoolError;
var
  header: PGpSharedPoolHeader;
  idx   : cardinal;
begin
  header := SafeGetHeader('FreeBuffer');
  idx := GetEntryIndex(buffer.Name);
  if idx = INVALID_HANDLE_VALUE then
    Result := speNotOwner
  else begin
    if Entry[idx].spieStatus <> Ord(iesAllocated) then
      raise Exception.Create('GpSharedMemory/TGpSharedPoolIndex.PrepareToSend:
        trying to send buffer with status '+IntToStr(Entry[idx].spieStatus));
    Entry[idx].spieStatus := Ord(iesQueued);
    Link(idx,header^.sphQueuedList);
    Result := speOK;
    FreeAndNil(buffer);
  end;
end; { TGpSharedPoolIndex.PrepareToSend }
```

➤ *Listing 8: Preparing the buffer to be sent back.*

At the first moment this may look like an addition specifically designed to break the data flow model. After all, this 'enhancement' allows you to pass any TGpSharedMemory to the SendBuffer, not just the buffers acquired from the pool. Although this is possible, it is not a recommended practice. This copy-on-send was designed to cope with something completely different: the restarted Reader problem.

Imagine the following, quite realistic, scenario. The Reader starts. After some time the Writer starts and acquires a buffer. The Reader then dies and is restarted (maybe automatically by a cluster manager or some monitoring service or process). The Writer sends the buffer (acquired from the previous Reader). What is happening behind the scenes?

When the Reader starts, its Initialize method creates two very special system primitives, a mutex named <user provided pool name>/Reader and a token (basically a mutex with some wrapping) named <user provided pool name>/Token/<automatically created GUID>. In other words, the Reader owns two global identifiers, one representing the Reader as a global service with a known name, the other representing this exact instance of the Reader. In other words, the Writer can verify the Reader's availability by checking the existence of the mutex, and it can verify that the Reader's identity hasn't changed by checking the existence of the token.

The Reader also creates an index, populates the header and directory, and starts listening to the message queue. The header contains the Reader's token. The allocated memory buffer in the directory (let's pretend there is only one for the sake of this discussion) is owner by the Reader. The name of this buffer is <Reader's token>/Pool/1.

Next, the Writer starts and creates its index object. The shared memory areas containing the index are now owned by two processes: Readers and Writers.

When the Writer calls AcquireBuffer, another shared memory object that uses the same mapping as <Reader's token>/ Pool/1 is created. The memory buffer (file mapping) is now owned by two processes.

After that, the Reader dies and is restarted. It recreates its mutex and token. The mutex has the same name as before, but the token doesn't. The Reader initialises a new index header, writes a new token inside, creates a new memory buffer and writes its name (<Reader's new token>/Pool/1) into the directory.

When the Reader died, the OS released its resources, one of which was the shared memory buffer. This buffer is (at that moment), owned only by the Writer. The newly created buffer is owned only by the Reader.

If the Writer was to free the old buffer at the moment (as happens at the end of the PrepareToSend method), Windows would destroy the file mapping containing the shared memory data, which is certainly not what we want to happen.

To prevent that, the code decides that the current Reader is not the owner of this shared memory (because the tokens in the shared memory name and index header doesn't match) and does the mumbo jumbo that copies the data into a Reader's shared memory buffer and... but I already told you that.

The net result is that the Writer doesn't care where the buffer came from, the old Reader or the new Reader. It will Send the buffer correctly in any case.

To complete this transaction, the Reader must somehow read the data from the buffer. Remember, the Writer posted the message WM_DATA_SENT to the Reader as the last action in the SendBuffer method. The reader receives this message in a slightly circuitous manner.

➤ *Listing 9: Processing received buffers.*

```
function TGpSharedPoolReader.ReadData: boolean;
var
  iData: integer;
begin
  if not AcquireIndex then
    Result := false
  else begin
    try
      Index.ReadData(StoreReceivedData);
    finally ReleaseIndex; end;
    if assigned(sprOnDataReceived) then begin
      for iData := 0 to sprReceivedDataList.Count-1 do
        DoDataReceived(TGpSharedMemory(sprReceivedDataList[iData]));
      sprReceivedDataList.Clear;
    end;
    //else wait for owner to read data explicitely via GetNextReceived
    SetEvent(sprDataReceivedEvent);
    Result := true;
  end;
end; { TGpSharedPoolReader.ReadData }

procedure TGpSharedPoolReader.StoreReceivedData(shm: TGpSharedMemory);
begin
  sprReceivedDataList.Add(shm);
  AcquiredList.Add(shm);
end; { TGpSharedPoolReader.StoreReceivedData }
```

```
function TGpSharedPoolIndex.TryToResize(numBuffers:
  cardinal): TGpSharedPoolError;
  function CreateMemory(idx: cardinal): boolean;
  begin
    try
      Entry[idx].spieHandle :=
        TGpSharedMemory.Create(GetEntryName(
        idx),BufferInitialSize,BufferMaxSize,false);
      Result := true;
    except
      Result := false;
    end;
  end; { CreateMemory }
var
  header: PGpSharedPoolHeader;
  iEntry: integer;
begin
  Result := speWin32Error;
  if not spiIsReader then
    raise Exception.Create('GpSharedMemory/
      TGpSharedPoolIndex.TryToResize: writer tried to resize
      shared memory pool');
  if not assigned(spiDirectoryBlock) then
    raise Exception.Create('GpSharedMemory/
      TGpSharedPoolIndex.TryToResize: directory does not
      exist');
  header := SafeGetHeader('TryToResize');
  if numBuffers = 0 then
    numBuffers := header^.sphResizeIncrement;
  while (numBuffers > 0) and (header^.sphDisposedList <>
    INVALID_HANDLE_VALUE) do begin
    iEntry := header^.sphDisposedList;
    Unlink(iEntry, header^.sphDisposedList);
    if not CreateMemory(iEntry) then
      Exit;
    Entry[iEntry].spieStatus := Ord(iesFree);
    Link(iEntry, header^.sphFreeList);
    Dec(numBuffers);
    Inc(header^.sphNumBuffers);
    Inc(header^.sphFreeBuffers);
  end; //while
  if (header^.sphNumEntries + numBuffers) >
    header^.sphMaxBuffers then
    numBuffers := header^.sphMaxBuffers -
      header^.sphNumEntries;
  if numBuffers > 0 then begin
    spiDirectoryBlock.Size := spiDirectoryBlock.Size +
      numBuffers*SizeOf(header^.sphIndexEntrySize);
    for iEntry := header^.sphNumEntries+numBuffers-1
      downto header^.sphNumEntries do begin
      if not CreateMemory(iEntry) then
        Exit;
      Entry[iEntry].spieStatus := Ord(iesFree);
      Link(iEntry, header^.sphFreeList);
    end; //for iEntry
    Inc(header^.sphNumBuffers, numBuffers);
    Inc(header^.sphFreeBuffers, numBuffers);
    Inc(header^.sphNumEntries, numBuffers);
  end;
  Result := speOK;
end; { TGpSharedPoolIndex.TryToResize }
```

➤ *Listing 10: Growing the pool.*

In the first version of the shared pool code, I simply created an internal window (AllocateHwnd). The message was sent to the window handle that was stored in the index header. As you know if you read the third part in the series, this doesn't work on newer Windows version if the code that created the window is in the service and the message sender is not. To solve that, I have created a messaging class that only uses shared memory and an event. The Writer now uses this messaging class (TGpMessageQueueWriter) instead of a simple PostMessage.

Because the internal window was handling some other messages that were created inside the Reader, I have decided to leave it there. I also made no changes in its internals, message handling code for this window still checks for the WM_DATA_SENT message and calls appropriate method. The new TGpMessageQueueReader merely receives WM_DATA_SENT via my own mechanism and reposts it to this internal window. This really is a plug-and-play solution to a complicated problem!

The main workhorse that triggers when WM_DATA_SENT is received is the TGpSharedPoolReader.ReadData method (Listing 9).

It uses a method from the index object, ReadData, to iterate over the buffers in the 'queued' list and a helper function, StoreReceivedData (also shown in Listing 9), called from ReadData. The former iterates over the 'queued' list and converts each entry back to 'allocated' status. It also creates a new shared memory object pointing to the same file mapping as the original one and passes this new object to the helper function (StoreReceivedData in our case). StoreReceivedData simply stores those objects into two internal lists: a list of acquired objects (so it can be freed on Destroy) and a list of received objects.

Those objects must now be passed to the external code somehow. If there is a handler for the OnDataReceived event defined, ReadData will for each received shared memory object, call this handler and then remove the object from the internal list. If the handler is not defined, ReadData only signals the DataReceivedEvent event (a public property). External code can wait on this event and call the GetNextReceived function in a loop. GetNextReceived returns the first object in the 'received' list, or nil if there is no such object. It also removes the object from the list.

Independent of the way the external code decides to read the data, synchronously (OnDatareceived) or asynchronously (DataReceivedEvent + GetNextReceived), it now owns that data (or, rather, the shared memory object that wraps the data). So it must, somewhere, destroy this object by calling the ReleaseBuffer method.

As a final thought, if the Writer decides that it doesn't want to send the buffer, it can simply call the ReleaseBuffer method, which moves the buffer back to the 'allocated' queue and destroys the Writer's shared memory object for this buffer.

Pool Sizing

Finally, let's take a look at how new buffers in the pool are allocated and how the 'sweeper' code works.

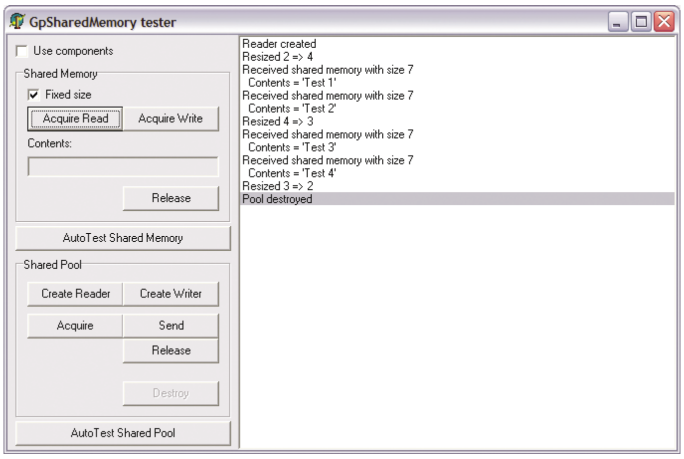
Resizing is done inside the method TGpSharedPoolIndex.TryToResize (Listing 10). After checking for some error conditions (indicating errors in the shared pool code), the main while loop iterates over the disposed list (containing directory entries with no memory buffer attached), moves entries to the free list and allocates shared memory objects.

Finally, this method resizes the directory shared memory object (if necessary) and initialises newly created entries.

Of course, during all that processing, the code makes sure that it doesn't allocate more buffers than allowed (resizeIncrement passed to TGpSharedPoolReader.Initialize). To release unnecessary buffers, the 'sweeper' (actually, TGpSharedPoolIndex.Sweep,

```
procedure TGpSharedPoolIndex.Sweep;
var
  header     : PGpSharedPoolHeader;
  idx        : cardinal;
  iEntry     : integer;
  iList      : integer;
  releaseList: TList{cardinal};
begin
  header := SafeGetHeader('Sweep');
  releaseList := TList.Create;
  try
    idx := header^.sphFreeList;
    while (idx <> INVALID_HANDLE_VALUE) and
      (header^.sphNumBuffers >
      (header^.sphMinBuffers+cardinal(
      releaseList.Count))) do begin
      if (Entry[idx].spieStatus = Ord(iesFree)) and
        ((Now-Entry[idx].spieReleasedAt)*SecsPerDay >
        header^.sphSweepTimeoutSec) then
        releaseList.Add(pointer(idx));
      idx := Entry[idx].spieNext;
    end; //while
    for iList := 0 to releaseList.Count-1 do begin
      iEntry := cardinal(releaseList[iList]);
      TGpSharedMemory(Entry[iEntry].spieHandle).Free;
      TGpSharedMemory(Entry[iEntry].spieHandle) := nil;
      Entry[iEntry].spieStatus := Ord(iesDisposed);
      Unlink(iEntry, header^.sphFreeList);
      Link(iEntry, header^.sphDisposedList);
      Dec(header^.sphNumBuffers);
      Dec(header^.sphFreeBuffers);
    end; //for
  finally FreeAndNil(releaseList); end;
end; { TGpSharedPoolIndex.Sweep }
```

➤ *Listing 11: Shrinking the pool.*



```
Reader created
Resized 2 => 4
Received shared memory with size 7
  Contents = 'Test 1'
Received shared memory with size 7
  Contents = 'Test 2'
Resized 4 => 3
Received shared memory with size 7
  Contents = 'Test 3'
Received shared memory with size 7
  Contents = 'Test 4'
Resized 3 => 2
Pool destroyed
```

➤ *Figure 2: Shared pool testbed.*

Listing 11) wakes up every 15 seconds (by default: you can change that), triggered by the internal `TTimer`.

The `Sweep` method walks over the free list and for each directory entry checks if the associated memory buffer has been released before the last pass of the sweeper. It adds all such entries into a temporary list.

When the first pass is done, the sweeper iterates over this temporary list and for each entry frees the associated shared memory object, changes the status to disposed, removes the entry from the free list and adds it to the disposed list. It also adjusts the global number of allocated buffers (`sphNumBuffers`) and the number of free buffers (`sphFreeBuffers`).

Example

This month's source code includes the testbed application that will help you explore the shared pool. Start two copies of the testGpSharedMem.exe program and click `CreateReader` in the first and `CreateWriter` in the second. Then play with the `Acquire`, `Send`, and `Release` buttons and watch the program's log (see Figure 2). Of course, you should also take a look at the main unit for this small test application, which is called `testGpSharedMem1`, to see how it uses the shared pool.

The testbed uses two interfaces to the shared pool: procedural (as described in the article) and componentized (which is just a light wrapper around the `TGpSharedPool*` classes). Of course you can use any of them in your application; you will know best which one suits you most.

Primoz Gabrijelcic is R&D Manager of FAB d.o.o. in Slovenia. You can contact him at gp@fab-online.com. All the code in this article is freeware and may be freely reused in your own applications.