Sharing Data With The Win32 API

by John Chaytor

This article describes an implementation of data sharing between Win32 processes running on Windows 95 or Windows NT which provides controlled access to common data and allows for notification of data updates. This has been implemented in two classes: `TEnque` which provides access control (via locking) and `TSharedMemory` which provides physical access to the data. It should be of interest to you if you want a rather straightforward method of sharing transient data between applications on a workstation. The `TEnque` object should be of interest to anyone needing to serialise access to any resource between processes.

Special versions of the demonstration programs and classes have been provided for Win16 (Delphi 1) applications which will run on Win95/NT (not Win32s). These Delphi 1 classes have the same functionality as the 32-bit versions. Data can be freely shared between 16/32-bit applications (including notification of update). The 64Kb segment limit does not apply.

Data Sharing Issues

Whenever there is a need to provide concurrent access to data, there are several areas which need to be addressed.

Data integrity. Whenever data is being updated by a thread, no other thread should be allowed update access to that data until the update is complete. If, for example, you allowed two programs free access to the same memory area there is a very high likelihood of data becoming corrupted. Concurrent update is definitely a no-no. But what about the situation where a thread is updating data and there are threads which need to read the data? Should the readers be locked out while the data is being updated or should they be allowed to read data while it is being updated? The answers are Yes and Yes! That is, there are no fixed rules. It depends upon the nature of the data. Usually, if the data is important or complex (eg business data which needs several pieces of information to be updated as a set before it is meaningful or valid) read access must be blocked until the data has been fully updated. However, if the data is not so important it is up to you, as a system designer, to decide if it is OK to read partially updated data. The important point is that you are aware of the nature (and implications) of the problem. The class described here caters for both approaches. Data access control is provided by `TEnque` class which makes use of `Mutex`, `Semaphore` and `Event` kernel objects.

Avoiding data duplication. Data sharing should not require duplication of data. This ensures that you only update data in one place. If you consider that the designers of Windows 95/NT have made great strides towards keeping applications apart you may think that it is extremely difficult to share data between applications. This is not the case. The designers have thought ahead and provided some pretty nice mechanisms to provide this capability. Using these techniques we only ever have one copy of the data on the workstation. This is provided by the `TSharedMemory` class which makes use of memory mapped files (which are not files in the traditional sense).

Performance. Data sharing should not cause the application to grind to a halt. As we are using facilities built into the operating system don't blame me if this is the case!

OnChange notification. When the data has been updated should we be notified? With the advent of data aware components it would seem criminal not to implement this capability. The class described has an `OnChange` event to allow the process to be informed whenever the data is changed by any process on the system.

Let's try... but finally.... The classes make extensive use of `try-finally` blocks. This is to ensure that we do not accidentally hold any resources. Once we venture into the inter-process arena we need to be extremely careful about releasing resources as soon as possible (and certainly when our application encounters a problem). If we fail to release a lock, we may cause other applications to fail or hang. If our program terminates this may not be a problem as Windows will usually clean up the resource usage and other applications will spring to life – however see the discussion of Semaphores later for a very nasty problem. The worst case scenario is when an application has an error, does not release a lock, but keeps running.

Resource Access With TEnque

The `TEnque` class has been designed to control access (ie control who is allowed to have read/update access at any particular time) to any named global resource on a workstation. It is called `TEnque` as the functionality of this object was provided on the IBM/MVS mainframe operating system many years ago by waiting on an Enque event. Who said mainframes have had their day?

The `TEnque` class provides for exclusive access (update in data terms) and shared access (read in data terms). It does this using three objects (`Mutex`, `Semaphore` and `Event`) which were introduced along with the Win32 API. These objects are system wide and are known as Kernel objects; this allows each process to have common knowledge of the state of these objects. There are several APIs available to create, destroy and change the state of these objects. Table 1 shows the API calls we will use to maintain access control, along with a brief description of how they are used. There are other APIs (and options) for these objects which are not discussed here. See the on-line help or the Win32 reference for further details.

A process is granted access to a Kernel object by making a `WaitForSingleObject` or `WaitForMultipleObjects` API call (from now on this will be referred to as `WaitFor...`). These calls will only return successfully if the objects are currently available. A process can make nested calls to these APIs and get, in effect, more than one lock on the resource. If this is done (for `Mutex` and `Semaphore` objects), there should be an equal number of calls to release the resource.

CreateMutex CreateSemaphore CreateEvent	When we call any of these three create APIs Windows will determine if the kernel object already exists and will create it if needed. If it already exists it simply passes a handle back to us. The name of each of these objects must be unique. For example, a `Mutex` and `Event` cannot have the same name. We cater for this by adding a suffix to the end of the name passed to the `TEnque` constructor.
WaitForSingleObject	Suspends the thread until the object is available.
WaitForMultipleObject	Suspends the thread until all the objects are available (all or nothing: as we use it).
ReleaseMutex	Indicates to Windows that we wish to relinquish ownership of the `Mutex`.
ReleaseSemaphore	Indicates to Windows that it should reduce the count by the amount specified. In this class we only ever reduce by one at a time.
ResetEvent	This is called whenever we have obtained read access to the resource. It indicates that readers are active and stops threads getting update access to the object (as they wait for the event to become available).
SetEvent	This is called when the last reader has finished and makes the event available. It indicates that update access can be granted.
CloseHandle	This is called for each kernel object in the destructor to free windows resources. Windows will release ownership for the object.

➤ *Table 1: Access control API calls*

Kernel Objects

When discussing the state of kernel objects the official term used to describe the current state is signalled or non-signalled. I find these terms confusing and difficult to remember so I tend to use available or not-available instead. Table 2 shows, for each kernel object, the circumstances under which the object is available (signalled).

Kernel Object	When Available (Signalled)
Mutex	The `Mutex` object is available as long as no thread currently has ownership. Ownership is granted via a successful `WaitFor...` call which causes the object to be marked as not-available (non-signalled) for other threads. When the thread calls `ReleaseMutex` it relinquishes ownership (subject to nested calls) and the state becomes available (signalled) once more.
Semaphore	The `Semaphore` object is available as long as the count is greater than zero. Each call to `WaitFor...` reduces the count. When it reaches zero it is marked as not-available. It remains in this state until a thread calls `ReleaseSemaphore` which increases the count.
Event	The `Event` object is available after a call to `SetEvent`. It remains available until a call is made to `ResetEvent`.

➤ *Table 2: Signalling of Kernel objects*

Mutex Object

`Mutex` is short for *mutually exclusive*. `Mutex` objects are ideal for situations where you need to get exclusive access to a resource. This is used in the `TEnque` class for two purposes. The first being to control update access to the user data (one at a time). The second is to get temporary control during critical routines within the class itself. For example, when releasing read access, we get ownership of the `Mutex` then determine if we are the last active reader. By getting ownership of the `Mutex` we ensure that all threads waiting on the object are locked out until we release it.

Semaphore Object

We will use a `Semaphore` object to keep a count of the number of readers. `Semaphore` objects are different to `Mutex` objects because they are not owned by any thread. Multiple processes can call the relevant APIs to increase or decrease the count (within the specified limit). For example, if a semaphore were created with a limit of 5, then five calls to the `WaitFor...` APIs (from any process, including multiple calls from the same process) would be allowed to decrease the count until it reached zero. If another `WaitFor...` call were made the task would be suspended until a `ReleaseSemaphore` call was made to increase the count. In the `TEnque` class the initial count is set to nearly 2 billion so this should never happen! That is, the `Semaphore` is always available. We use the `Semaphore` simply as a counter for the number of readers.

One problem with `Semaphore` objects is that there is no easy way to know the current count which makes it difficult to determine if a process can get update access. We control update access indirectly by the use of an event object. A second and more serious problem is that Windows does not clean up the Semaphore count if an application aborts: see later...

Event Object

Event objects, not surprisingly, are used to signal when some event has occurred. They can be either auto-reset or manual-reset objects. For an auto-reset Event, when a thread successfully calls WaitFor... Windows changes the state of the Event back to not-available. A subsequent call to WaitFor... would not return successfully until SetEvent was called again. For manual-reset Events Windows does not change the state of the Event after the WaitFor... call. This means that multiple threads will be released without the need to call SetEvent again. The manual-reset Event is exactly what we need. We call SetEvent when the last reader ends and the Event must remain available until ResetEvent is called. Any updater making a request will find that the Event is available. Whenever a reader starts a call is made to ResetEvent to mark the Event as not-available to inhibit the granting of update access.

TEnque: Creating The Object

Listing 1 shows a functional version of the constructor for the TEnque class. The constructor accepts a parameter which will be used as the base name for the object. To ensure that the three kernel objects have unique names we add a suffix for each object type (_M, _S, _E).

To discuss the API calls used to create these objects we need to look at the declaration for each. They are found in WINDOWS.PAS. For each of these API calls the first parameter is a pointer to a security attributes structure. As this is beyond the scope of this article we pass nil to accept the default security descriptor provided by Windows. The final parameter (lpName) is the name to be used to create the object (each with its own suffix). Table 3 describes the key points of the three functions.

```
function CreateMutex(lpMutexAttributes: PSecurityAttributes;
  bInitialOwner: BOOL; lpName: PChar): THandle; stdcall;
```

The second parameter (bInitialOwner) indicates if we wish to be granted immediate ownership of the Mutex as it is being created (this would block other access). We pass False as ownership will be requested explicitly.

```
function CreateSemaphore(lpSemaphoreAttributes:
  PSecurityAttributes; lInitialCount, lMaximumCount: Longint;
  lpName: PChar): THandle; stdcall;
```

The third parameter (lMaximumCount) is the maximum count allowed for the semaphore. As we will allow any number of readers we set this figure to $7FFFFFFF (2 billion minus 1!). The second parameter (lInitialCount) indicates the initial count for the Semaphore. This must be between 0 and lMaximumCount. As we are going to allow for an unlimited number of readers this is also set to $7FFFFFFF. Every call to WaitFor... will decrease the count. As there is no possibility of the count reaching 0 (at least on my PC!) the Semaphore will never be the cause of a delay waiting for an object. Once a reader has finished it calls ReleaseSempahore which increases the count. When the count reaches $7FFFFFFF again we know that this is the last reader running. The importance of this fact is covered in the section *Controlling Read Access*.

```
function CreateEvent(lpEventAttributes: PSecurityAttributes;
  bManualReset, bInitialState: BOOL; lpName: PChar): THandle;
  stdcall;
```

The second parameter (bManualReset) indicates if we want to create a manual reset Event. We pass True as this is indeed what we are going to do. False would indicate an auto reset Event. The third parameter indicates the initial state for the object. We pass True as we want the initial state to be available.

➤ *Table 3*

```
constructor TEnque.Create(Name: string);
var WrkName: array[0..255] of char;
begin
 { Ensure there is enough room for required suffix }
  FName := Copy(Name,1,253);
  FExclusiveAccess := CreateMutex(nil, FALSE,
    StrPCopy(WrkName, FName+MUTEX_SUFFIX));
  if FExclusiveAccess = 0 then
    Raise Exception.CreateFmt(
      'Error creating the Mutex object. RC was %d',[GetLastError]);
  FNumberOfReaders :=  CreateSemaphore(nil,MAX_SEMAPHORES,MAX_SEMAPHORES,
    StrPCopy(WrkName,FName+SEMAPHORE_SUFFIX));
  if FNumberOfReaders = 0 then
    Raise Exception.CreateFmt(
      'Error creating the Semaphore object. RC was %d',[GetLastError]);
  FReadersActive := CreateEvent(nil, TRUE, TRUE,
    StrPCopy(WrkName, FName+EVENT_SUFFIX));
  if FReadersActive = 0 then
    Raise Exception.CreateFmt(
      'Error creating the Event object. RC was %d',[GetLastError]);
end;
```

➤ *Listing 1*

Controlling Update Access

Listing 2 shows the two methods which get and release update access to the resource. Before we can get update access to the object, both the Mutex and Event objects must be available. The Mutex object ensures that we cannot get access when another thread owns the Mutex, and the Event object will only be available if there are no readers active. So, we wait for both objects using the WaitForMultipleObjects call. If either object is not available Windows will suspend the thread until they are both available. When the API returns we can accept a return value of WAIT_OBJECT_0 or WAIT_ABANDONED – this latter value is a special case which means that the thread which previously owned the Mutex terminated without releasing the object cleanly.

Before we do this, a check is first made to ensure that we do not currently have read access and an error is generated if this is the case. This restriction is required as, when readers are active, the event is not available. The WaitForMultipleObjects would not return in this situation and the process would be suspended. This is not as bad as it seems as write access implies read access. It just means that you cannot change from read access straight to update access. You would need to release read access first then get update access.

Releasing update access is very straightforward. By definition, we already have ownership of the Mutex object at this point so we simply call ReleaseMutex to mark it available (note: we don't do anything with the Event object here as the status of the Event is controlled by the number of readers, the WaitForMultipleObjects call in GetUpdateAccess did not alter the state of the Event object).

Controlling Read Access

Listing 3 shows functional versions of the two methods which get and release read access to the resource along with the Lock and Unlock methods used in the method ReleaseReadAccess.

To get read access we need to wait for the Mutex and Semaphore objects. As the Semaphore is simply a counter (with no practical limit in our case) we are really only waiting

➤ *Listing 2*

```
function TEnque.GetUpdateAccess: Boolean;
var WaitResult: LongInt;
    WaitObjects: array[0..1] of LongInt;
begin
  if FReadlocks > 0 then
    Raise Exception.Create(
     'Cannot grant update access while have read access ')
  else begin
    WaitObjects[0] := FExclusiveAccess;
    WaitObjects[1] := FReadersActive;
    WaitResult := WaitForMultipleObjects(2,@WaitObjects,TRUE,FWriteTimeout);
    case Waitresult of
      WAIT_OBJECT_0,WAIT_ABANDONED:
        begin
          Inc(FUpdateLocks);
          SetStatusString(ENQ_OK);
          Result := True;
        end;
      WAIT_TIMEOUT:
        Raise Exception.Create('Timed out waiting to get update access');
    else
      Raise Exception.CreateFmt(
        'Got error %d while waiting for update access',[GetLastError]);
    end;
  end;
end;
procedure TEnque.ReleaseUpdateAccess;
begin
   if FUpdateLocks > 0 then begin
     Dec(FUpdateLocks);
     ReleaseMutex(FExclusiveAccess);
   end else
     Raise Exception.Create(
      'Invalid call to ReleaseUpdateAccess - no current update access');
 end;
```

➤ *Listing 3*

```
function TEnque.GetReadAccess: Boolean;
var WaitResult: LongInt;
    WaitObjects: array[0..1] of LongInt;
begin
  WaitObjects[0] := FExclusiveAccess;
  WaitObjects[1] := FNumberOfReaders;
  WaitResult := WaitForMultipleObjects(2, @WaitObjects,
    TRUE, FReadTimeout);
  case WaitResult of
    WAIT_OBJECT_0,WAIT_ABANDONED:
      begin
        ResetEvent(FReadersActive);
        Inc(FReadLocks);
        ReleaseMutex(FExclusiveAccess);
        Result := True;
      end;
    WAIT_TIMEOUT:
      Raise Exception.Create(
        'Timed out waiting for read access');
  else
      Raise Exception.CreateFmt('Got error %d when'+
        ' waiting for read access',[GetLastError]);
  end;
end;
function TEnque.ReleaseReadAccess: Boolean;
var PreviousCount: LongInt;
begin
  Result := False;
  if FReadLocks = 0 then
    Raise Exception.Create('No current read access');
  else begin
    { Lock ensures we get accurate value for number of readers }
    Lock;
    try
      Dec(FReadLocks);
      ReleaseSemaphore(FNumberOfReaders, 1,
        @PreviousCount);
      { If this was the last reader, mark event as
        available }
      if PreviousCount = $7FFFFFFE then
        SetEvent(FReadersActive);
      Result := True;
    finally
      UnLock;
    end;
  end;
end;
procedure TEnque.Lock;
var WaitResult: LongInt;
begin
  WaitResult := WaitForSingleObject(FExclusiveAccess,
    Max_Maintenance_Delay_Expected);
  case WaitResult of
    WAIT_OBJECT_0,WAIT_ABANDONED:
      ; { We are OK }
  else
    Raise Exception.Create(
      'Failed to get lock! THIS SHOULD NOT HAPPEN!');
  end;
end;
procedure TEnque.UnLock;
begin
  ReleaseMutex(FExclusiveAccess);
end;
```

for the Mutex. Once we get access to this, the first thing we do is call ResetEvent for the Event object. This marks the object as not available (ie readers are active). This means that no update access request will be satisfied until a SetEvent is called (described in the next paragraph). We then release the Mutex as we only needed to get ownership of it to ensure that we call the ResetEvent before any updaters get ownership (remember, they wait for the Mutex also).

Releasing read access is the most critical part of the whole process. We need to ensure that we can accurately detect that the last reader has ended as we need to call SetEvent at that time (to indicate there are no readers active) We cannot do this by simply calling ReleaseSemaphore, checking if we were the last reader then calling SetEvent. Due to the pre-emptive nature of Windows 95/NT, between these calls other processes may get ownership of objects and change the situation. We get round this by calling the Lock method, which gets access to the Mutex object, ensuring that we have total control. We then call ReleaseSemaphore which will increase the count by one. ReleaseSemaphore accepts a var parameter (the last parameter) which will be set to the count before it was increased. If this value is $7FFFFFFE we know this is the last reader ending and call SetEvent. We then call the UnLock method to release the Mutex.

Note that in the Lock method the wait delay is not the standard timeout value (which could be infinite). It is set to a small value as, at this time, there should be no thread holding an extended lock on the Mutex object. However, we cannot pass 0 as it is possible for multiple processes to be calling ReleaseReadAccess at the same time, so there is a possibility of a time-out being returned if we pass 0. If we do time out an exception is raised, indicating something is wrong!

Enough Theory, Lets Play...

Now that we understand how to control access to a resource we can now play with a simple demo. The NQDEMO.EXE application allows you to request read and write locks to a named resource to visually see how the applications wait for the resource to become available. The listbox shows an activity log for each instance. Figure 1 shows the single window which is displayed when you run the program. You will need to run multiple instances of the program. In each instance, you should request read or write access. You will see that, while an instance has read access, other instances can also get read access. However, no update access will be granted: they will wait until all readers have released their locks. Similarly, if you request update access, no other instance will be granted any access until the update lock has been released.

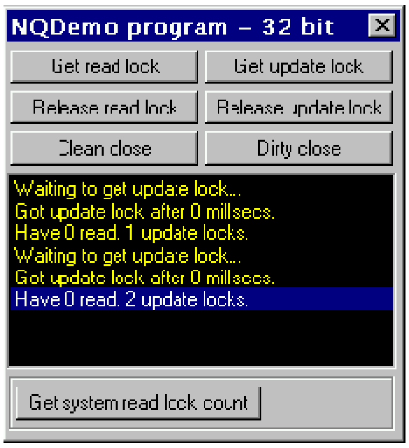
A restriction on the TEnque class is that, for a thread, it is invalid to be granted update access if you already have read access (as described previously). If you attempt this an exception will be generated.

A good demonstration to show inter-process locking is to request read access for an instance, then request update access in all the other instances (they will be suspended). When you release the read access only one of the waiting instances will be granted update access. As you then release each lock, the instances will be activated in turn.

If you close the instance without explicitly releasing the locks (via the Dirty close button) you will see that the other instances spring to life. If you look at the code for this program (on the disk of course) you will see that the Enque object is not freed when you click this button. This demonstrates Windows is cleaning up the resources.

Semaphore Blues...

That last sentence is not strictly correct. Windows does not clean up a Semaphore object correctly if the application terminated without calling ReleaseSemaphore. In the NQDEMO application, if you get read locks then click the Dirty close buttons you will find that no thread will be granted update access. This seems to be because Windows (even NT) does not keep track of the count allocated for a particular thread. If, for example, an instance of NQDEMO has 3 read locks and then it does not perform a clean close, the Semaphore count will not be decreased when the program ends! Try this for yourself. For one of the instances get several read locks. Click the Get system read lock count button to display the number of reader locks. Then end the program using the Dirty close button. In another instance click the button again: the count will be the same. This means that SetEvent will never be called as the count for the Semaphore is now invalid. Therefore, in applications, it is essential that the TEnque destructor is called to clean up. If this situation ever arises it can only be fixed by closing all applications to destroy the Semaphore object.



```
Waiting to get update lock...
Got update lock after 0 millsecs.
Have 0 read. 1 update locks.
Waiting to get update lock...
Got update lock after 0 millsecs.
Have 0 read. 2 update locks.
```

➤ *Figure 1*

Data Access With TSharedMemory

This class provides physical access to the shared data by using memory mapped files. It makes use of the TEnque class to co-ordinate the timing of access.

Win32 supports an object called a memory mapped file. As you would expect, there are various tricks you can do with memory mapped files. This section details how they are used in this class. The Delphi on-line help and Win32 API reference have more information.

A memory mapped file is one mechanism by which data can be shared between processes. All process see the same data. When we create the object Windows allocates space in the swap file for the file. Once it has been created, you can simply access it as if it were any other memory allocated in your application. It is that simple! Table 4 shows all the APIs we use to maintain memory mapped files along with a short description for each.

When discussing this class, there are six areas which need to be covered: creating the object, internal control data and processes, data access co-ordination, mechanism for the notification of update, destroying the object, and finally reading without integrity.

Creating The Object

Listing 4 shows the constructor for the `TSharedMemory` class. The `TSharedMemory` class used in the demo applications creates and maintains a single dimension array. The constructor has three parameters: the name, element size and capacity of the array. All instances which create the `TSharedMemory` object must specify the same values. This is not due to a restriction of memory mapped files, it simply allows for the basic functionality of memory mapped files to be demonstrated without getting in too deep!

Before we create the memory mapped file we create the `TEnque` object and get exclusive access to it. This is to ensure that while the constructor executes we have total control.

After creating and getting update access to the `TEnque` object we create the two memory mapped files. For the `CreateFileMapping` API we indicate that the type will be a memory mapped file, we want read/write access, and specify the size for each. When we have successfully created the `Ctrl` memory

➤ *Table 4*

CreateFileMapping	We pass a name, type (memory mapped file), size and access mode (write) to create the object. We get a handle back for the object.
MapViewOfFile	This maps the requested part of the file mapping object (in our case the whole file) into the process. The API returns the address of the start of the file in your process. Once this has returned, the data in the file can be accessed like any other memory allocated by your process: you even get access violations if you go past the end!
UnMapViewOfFile	This can be thought of as de-allocating the memory. Once this has been called it is invalid to try and access the memory.
CloseHandle	De-allocates the memory mapped file from the process.

➤ *Listing 4*

```
constructor TSharedMemory.Create(Name: string;
Element_Size,Capacity: LongInt);
var
  WrkName: array[0..255] of char;
  FirstInstance: Boolean;
  MName: string;
begin
  { Truncate name to allow for required extensions }
  MName := Copy(Name,1,249);
  { Create the TEnque object to control access to the
user and control data }
  StrPCopy(WrkName,MName);
  FEnque := TEnque.Create(StrPas(WrkName));
  if not Assigned(FEnque) then
    Raise Exception.CreateFmt('Failed to create TEnque
object for %s',[MName]);
  { Lock out all processes until we initialise }
  FEnque.GetUpdateAccess;
  try
    { Create the 'Ctrl' memory mapped area which is
private to this class and maintained exclusively by this
class }
    StrPCopy(WrkName,MName + 'Ctrl');
    FCtrlFileMapHandle :=
CreateFileMapping(MEMORY_MAPPED_FILE,nil,PAGE_READWRITE,0,
SizeOf(TSMCtrlDetails),WrkName);
    if FCtrlFileMapHandle <> 0 then begin
      FirstInstance := GetLastError <>
ERROR_ALREADY_EXISTS;
      FCtrlAddress :=
MapViewOfFile(FCtrlFileMapHandle,FILE_MAP_WRITE,0,0,0);
      if FCtrlAddress = nil then
        Raise Exception.CreateFmt('Failed to create view
of the control file mapping object
%s',[StrPas(WrkName)]);
      if FirstInstance then begin
        if Capacity < 1 then
          Raise Exception.CreateFmt('Capacity invalid
for new TSharedMemory object (%d)',[Capacity]);
        if Element_Size < 1 then
          Raise Exception.CreateFmt('Element_Size
invalid for new TSharedMemory object
(%d)',[Element_Size]);
        Fillchar(FCtrlAddress^,sizeof(TSMCtrlDetails),0);
        with FCtrlAddress^ do begin
          Ctrl_Element_Size := Element_Size;
          Ctrl_Capacity := Capacity;
        end;
      end else begin
        if FCtrlAddress.Ctrl_Capacity <> Capacity then
          Raise Exception.CreateFmt('Capacity invalid
for TSharedMemory object. Was %d, should be %d',
[Capacity, FCtrlAddress.Ctrl_Capacity]);
        if FCtrlAddress.Ctrl_Element_Size <>
Element_Size then
          Raise Exception.CreateFmt('Element size
invalid for TSharedMemory object. Was %d, should be %d',
[Element_Size, FCtrlAddress.Ctrl_Element_Size]);
      end;
    end else
      Raise Exception.CreateFmt('Failed to create
control file mapping object %s', [StrPas(WrkName)]);
    { Create the 'User' memory mapped area which can be
accessed by any user application }
    StrPCopy(WrkName,MName + 'User');
    FUserFileMapHandle :=
CreateFileMapping(MEMORY_MAPPED_FILE,nil,PAGE_READWRITE,0,
Element_Size * Capacity,WrkName);
    if FUserFileMapHandle <> 0 then begin
      FUserAddress :=
MapViewOfFile(FUserFileMapHandle,FILE_MAP_WRITE,0,0,0);
      if FUserAddress = nil then
        Raise Exception.CreateFmt('Failed to create view
of the file mapping object %s',[MName]);
    end else
      Raise Exception.CreateFmt('Failed to create file
mapping object %s',[MName]);
  finally
    FEnque.ReleaseUpdateAccess;
  end;
end;
```

mapped file we call GetLastError: not because we had an error, it is used to find out if the memory mapped file has already been created by another process. If GetLastError does not return ERROR_ALREADY_EXISTS we are the first instance to create the object and can initialise the Ctrl area. The base addresses of both memory mapped files are stored in instance variables. Once we have created the memory mapped files we release the lock on the TEnque object.

```
type
  PSMCtrlDetails = ^TSMCtrlDetails;
  TSMCtrlDetails = packed record
    Ctrl_Element_Size: LongInt;
    Ctrl_Capacity: LongInt;
    Ctrl_Count: LongInt;
    Ctrl_Data_Updated: Boolean;
    Ctrl_Num_Windows: LongInt;
    Ctrl_WinHandles: array[1..Max_OnChange_Notifications] of LongInt;
    Ctrl_NotificationBeenPosted:
      array[1..Max_OnChange_Notifications] of Boolean;
    Ctrl_Update_Count: LongInt;
  end;
```

➤ *Listing 5*

Internal Control Data And Processes

Listing 5 shows the structure of the Ctrl area. The Ctrl_Element_Size and Ctrl_Capacity fields are used to check that each thread which creates the object specifies the same array size and capacity. If a mis-match occurs the constructor raises an exception. The Ctrl_Count contains the current number of entries in the array. Ctrl_Data_Updated is used to indicate if a thread actually performed an update. Whenever the first update lock has been obtained this flag is set to False, then, whenever an update call (Add, Put, Reset) is made, the flag is set to True. This is used to determine if processes need to be notified when the lock is released. The Ctrl_Num_Windows, Ctrl_WinHandles, Ctrl_NotificationBeenPosted and Ctrl_Update_Count are discussed later when we cover the mechanism for the notification of update.

Data Access Co-Ordination

The TSharedMemory class provides GetReadAccess and GetUpdateAccess along with ReleaseReadAccess and ReleaseUpdateAccess to allow an application to explicitly control access. If you look at the implementation for all methods which access the data (Add, Get, Put, Reset) you will see that they check to see if the thread already has the required access and call the relevant access routine to ensure that they are authorised to access the data if this is not the case.

The Get/Release access routines have been provided to allow a program to obtain access (either read or write) for multiple data access calls. If you want to make more than one data access call, and lock all other processes out while you are doing so (a classic case is if you want to implement a rollback facility in the case of error), call the Get[Read/Write]Access method first, make all your calls to the data access methods, then call Release[Read/Write]Access to allow other processes to obtain access. If you do not need to hold locks between individual data access calls there is no need to call these routines.

Update Notification

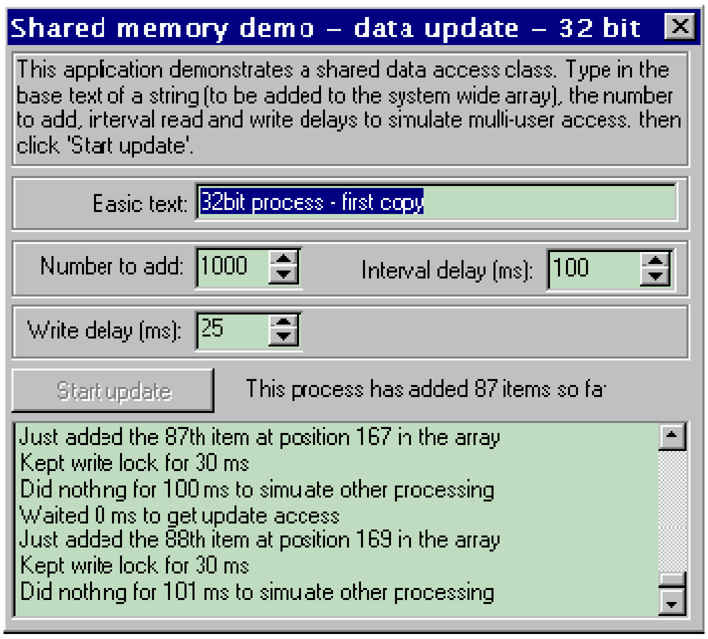
Notification of update has been implemented by posting messages. This means that the updating thread will not be held up. Whenever an OnChange event is set, the class creates a window using AllocHWnd (its WndProc is a method in the class) and stores the window handle in the Ctrl_WinHandles array. Then, whenever a process has updated the data, a message is posted to all windows just before releasing the last update lock. To avoid flooding the system with multiple messages (the target application may be busy doing other work) the Ctrl_NotificationBeenPosted array is used to indicate if a message has already been posted. Thus, a duplicate message will not be sent if this flag is set.

When the WndProc method processes the message it will clear this array value to indicate that it has processed the update. As posted messages are received asynchronously, we may get a message whilst we are updating ourselves. If this happens, we don't call the OnChange event, but we do clear the flag: when we finish updating we will be posted again.

A problem I encountered was that some OnChange notifications were not being processed in the 16-bit version when I stressed the system by running multiple programs. This was a timing problem which meant that, after calling the FOnChange event in the WndProc method an update was being performed in another thread before I reset the flag. Hence the update was not posted. As I wanted to keep the 32/16-bit versions compatible I had to implement a technique that would work in the 16-bit version. After exploring various options I decided to implement a simple counter field called Ctrl_Update_Count in the Ctrl area. This counter is incremented every time the NotifyDataChanged method is called and is passed as the lParam parameter in the PostMessage. Then, when the message is processed, a comparison is made between this and the current value of Ctrl_Update_Count. If they are different the WndProc posts a new message to itself with the new count in lParam which will trigger the OnChange event again. Only when the counts match is the Ctrl_NotificationBeenPosted flag cleared. If there was not a requirement to be compatible with 16-bit apps this could be removed. However, the overhead of this processing is negligible.

Destroying The Object

Destroying this object is simplicity in itself. All we need to do is call UnMapViewOfFile and CloseHandle



Shared memory demo – data update – 32 bit

This application demonstrates a shared data access class. Type in the base text of a string (to be added to the system wide array), the number to add, interval read and write delays to simulate multi-user access, then click 'Start update'.

Basic text: 32bit process - first copy

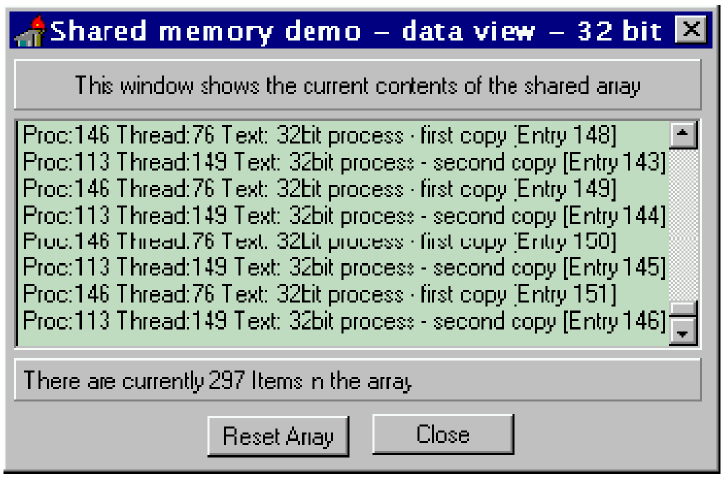
Number to add: 1000   Interval delay (ms): 100

Write delay (ms): 25

This process has added 87 items so far

```
Just added the 87th item at position 167 in the array
Kept write lock for 30 ms
Did nothing for 100 ms to simulate other processing
Waited 0 ms to get update access
Just added the 88th item at position 169 in the array
Kept write lock for 30 ms
Did nothing for 101 ms to simulate other processing
```

➤ *Figure 3*



Shared memory demo – data view – 32 bit

This window shows the current contents of the shared array

```
Proc:146 Thread:76 Text: 32bit process - first copy [Entry 148]
Proc:113 Thread:149 Text: 32bit process - second copy [Entry 143]
Proc:146 Thread:76 Text: 32bit process - first copy [Entry 149]
Proc:113 Thread:149 Text: 32bit process - second copy [Entry 144]
Proc:146 Thread:76 Text: 32bit process - first copy [Entry 150]
Proc:113 Thread:149 Text: 32bit process - second copy [Entry 145]
Proc:146 Thread:76 Text: 32bit process - first copy [Entry 151]
Proc:113 Thread:149 Text: 32bit process - second copy [Entry 146]
```

There are currently 297 Items in the array

➤ *Figure 3*

for each of the memory mapped files. We then destroy the TEnque object.

Reading Without Integrity

The class provides a boolean property called ReadWithoutIntegrity which, if set to True, will cause the class to always allow read access regardless of the state of the synchronisation objects: no reference to the TEnque object is made. A single restriction imposed is that this value cannot be set if you currently have read locks.

Demo Applications

Along with NQDEMO described previously, two other programs have been provided on the disk: SMTEST and SMSHOW. Figures 2 and 3 show example forms during execution.

To demonstrate data sharing you should run at least two copies of SMTEST and at least one copy of SMSHOW. The SMTEST program simply adds entries (containing the process id, thread id and text) to an array and the SMSHOW application is notified when the data has changed via the OnChange event: it simply displays the current contents of the array and the total number of entries.

In SMTEST, you should enter some basic text to be added to the array (the entry number will be appended), the number of entries that this instance should add, the write delay (ie how long to hold the update lock) and an interval delay (to simulate other processing). When you click the Start update button the program will simply loop round and add entries to the array, obtaining/releasing locks as required.

If you have more than one copy of SMTEST running you will see that the entries in the array contain items added by each instance. The activity log at the bottom of the form shows how long the process waited to get a lock, a line to indicate where it added the text to the array, how long it kept the lock, and finally how long it went to sleep and did nothing.

If you have multiple copies of SMSHOW running you will see that they are all updated whenever the data is changed. The SMSHOW application also has a button Reset Array. If you click this you will see that the array contents are reset.

If you also run the NQDEMO program while SMTEST and SMSHOW are running you will see that, if you obtain a lock in NQDEMO, the SMTEST program is suspended until you release the lock.

Also, if you use the NQDEMO program to get a lock then attempt to start an instance of SMTEST it will appear that the program has not started. This is because initialisation is done in FormCreate and the constructor for the TSharedMemory class waits to get exclusive access to the Mutex object. Once you release the lock the SMTEST form will appear (in a real application you would not do this initialisation in the FormCreate for this reason).

The demo programs specify an array containing 2000 elements to ensure that the total size is greater than the 16-bit 64Kb segment limit.

Changes For Delphi 1

16-bit versions of the two classes described here have been created. This section details the issues raised during this exercise, some of which certainly educated me! Rather than create common units with loads of $IFDEFs I have created 16-bit versions of the units for the classes (ENQUE16, SHAREM16) and demo apps (NQDEMO16, SMTEST16, SMSHOW16) to hopefully simplify things.

To call the Win32 APIs in Delphi1 it is necessary to perform thunking calls to the APIs in KERNEL32.DLL and USER32.DLL. Thankfully, as this is now common practice, a mechanism to do this has been released to the public domain. I used a unit provided by Christian Ghisler. The unit he created, CALL32NT, has been used to do the thunking. See the source for this unit on the disk as it describes how this is done. I have made a single change to the Call32NT unit: I made the procedure XLatHWnd public as I needed to make use of it (see later).

Basically, for each API you wish to use, you call a function called Declare32 to specify the API you are going to call and pass the argument types. It stores this information internally to know how to convert your 16-bit parameters to 32-bit parameters and returns a unique id back to you.

For each API you intend to call, you need to define a procedure type variable. The procedure type specifies the expected parameters, along with an extra parameter at the end which is the id returned from the call to Declare32. Each of these variables are set to the address of the function Call32. Therefore, Call32 is called for all the thunk calls. It uses the id to determine how to convert the parameters on the stack. It calls the relevant Win32 API, then, on return, amends the stack so that we can access any function return values. See the unit THUNK_32 where I do all initialisation.

There are certain constants used in the APIs we are calling which are defined in the Delphi 2 unit WINDOWS.PAS but not declared in Delphi 1. I have created these constants in unit WIN32DEF.PAS on the disk. Where there were name clashes with WINDOWS.PAS I have changed the name slightly in the new unit.

You will see that several APIs either return 32-bit flat addresses or expect them as parameters. As we cannot use the Pointer type (as this is the beloved segment style in Delphi 1) they are defined as a LongInt, which is the same length. Thus in the code, after making a call to a Win32 API which returns an address, we compare the result to 0 rather then nil (it has the same effect though).

Finally, something which caused me a few problems was that the size of the Ctrl area was different in Delphi 1 and Delphi 2: this was caused by the compiler packing the structures differently, so when I was attempting to run 16-bit and 32-bit apps at the same time I was getting very strange effects. Once I identified the problem I added the packed keyword to the TSMCtrlDetails and TArray_Element record structures to avoid this.

TEnque For Delphi 1

The Delphi 1 version of TEnque is found in ENQUE16.PAS and porting it was the first complication I ran into. When a WaitFor...Object call was made, the whole address space was suspended if the objects were not currently available. This meant that all my other 16-bit apps (including Delphi) were frozen out until the Wait... call completed. If the lock was being held by another 16-bit app (which it was) it could not release the lock as the address space was suspended... Well, Ctrl+Alt+Del fixed that particular problem!

After first suspecting that it was a Windows 95 problem I did some testing on NT which showed that this was also the case for 16-bit apps on NT. Although synchronisation objects (such as a Mutexes) are supposed to be associated with a thread this seems not to be the case if you attempt to use these objects from 16-bit apps. I'm not surprised really. I find it amazing that so much can work via thunking anyway! Although this could be circumvented on NT by running each 16-bit application in its own address space this is a bit over the top.

After a bit of head scratching I then hit upon the very simple idea of calling the WaitFor... APIs in a loop (which would give up after the required time-out period), specifying 0 as the time-out value and checking the result to see if we got control of the objects. If we didn't I put the process to sleep for 50 milliseconds (via the Win32 Sleep API) then tried again. Bingo! It worked. The performance monitor on NT 4.0 showed less than 2% CPU usage with eight 16-bit apps looping to get a read lock which was not currently available.

TSharedMemory For Delphi 1

The two areas which complicated things when porting this class over to Delphi 1 were firstly, that 16-bit window handles (as used in the NotifyDataChanged method) are not compatible with Win32, and secondly the difference between the 32-bit flat memory architecture and the old segmented 16-bit architecture.

The first problem was easily solved. The Call32NT unit has a private procedure called XlatHwnd which determines the 32-bit handle for a given 16-bit window handle. I simply made it public by defining the function in the interface section. Thus, in the Ctrl array, I stored the 32-bit handle. If a 32-bit application updates the data, Windows will send the message to our 16-bit WndProc. When a 16-bit application needs to notify that an update has been made, it simply calls the 32-bit version of PostMessage. This ensures that all WndProcs (including the 16-bit ones) will get the messages. The original 16-bit handle is stored to allow it to be destroyed.

The fact that both the 16-bit and 32-bit versions of this class are dealing with the 32-bit flat memory architecture has proved to be less of a problem than I originally expected.

If you recall, the TSharedMemory constructor creates two memory mapped files: a small Ctrl file which contains internal information, and a second file for the real user data. The user data is not a problem at all as it is only ever copied in or out via calls to the 32-bit API MoveMemory (for details see the additional comments in the file THUNK_32.PAS on the disk). The TSharedMemory class never uses or creates a 16-bit pointer to this data. It has no need to. However, the Ctrl area is unfortunately a different matter. This is accessed frequently within the TSharedMemory class via the FCtrlAddress pointer. Therefore, once we have created the memory mapped file we need to convert the flat 32-bit address to a 16-bit version. This is done using the function Convert32BitAddrTo16.

Observations On The 16-Bit Version

There is no doubt that 16-bit applications get a raw deal when it comes to obtaining access to the synchronisation objects. I ran a test with four 32-bit versions of SMTEST, four 16-bit versions of SMTEST, four 32-bit versions of SMSHOW, four 16-bit versions of SMSHOW, one 32-bit version of NQDEMO, and one 16-bit version of NQDEMO.

The 16-bit SMTEST programs were updating every 100 milliseconds, the 32-bit versions event 500 milliseconds. The 32-bit copies of SMTEST were running at approximately 5 times the rate of the 16-bit versions. I partly suspect that this is caused by the sheer processing involved in the `OnChange` event handler, effectively locking up the 16-bit address space. For example, when I clicked the `Reset array` button for a 32-bit app, the array was reset almost immediately. However, for the 16-bit apps, it sometimes took 10-15 seconds for the array to be reset. Admittedly this was on a totally stressed system *[Sounds just like I feel... Editor]*.

Finally...

The demonstration programs and classes described in this article only touch the surface of this whole area. To give these topics justice would require several chapters of a book.

In fact, the excellent *Advanced Windows* by Jeffrey Richter does exactly that! The `TEnque` object was created after reading his discussion of synchronisation objects, although I have approached the last reader problem in a different way.

This article barely touches on memory mapped files. Due to limited space I had to implement a simple version which just allocated a fixed file size. Don't let this put you off. For example, a different class I created dynamically grows the memory mapped file as you add more data (as a standard file would). There is no restriction to the initial size. Again, if you want to pursue this area the *Advanced Windows* book will provide you with all the information required. The Win32 API reference, or numerous articles on Microsoft's MSDN CD-ROM, are also a good source of information.

I hope you found this article informative and useful. Obviously, if you want to create a new class with different functionality you can totally redesign the classes. They were created to simply provide and example of memory mapped files and synchronisation objects and how they could interact.

John Chaytor is a freelance programmer who lives and works in Brighton, UK, and can be contacted via Compuserve as 100265,3642