A Synchronisation Toolkit

by Primoz Gabrijelcic

Every programmer accumulates reusable code over the years and I am no exception. Sometimes, parts of that code seem generic enough to be worth documenting and presenting it to the public.

Today, I'd like to put on display four classes belonging to the inter-process synchronisation part of my personal toolbox. You can start using them right now.

This article is the first in a short occasional series (in other words, they won't necessarily follow on month-by-month). The final goal is to present a set of classes that can broadcast 'events' between applications on one computer (something like Delphi's event model except that it supports multiple processes and multiple listeners). We will reach that goal in a few stages, but first we will take a look at some very basic synchronisation tools. Then we will build a powerful shared memory object, match it with XML, and finally start broadcasting events around.

Finally, a warning. This is not an entry-level article. I would recommend that you be at least acquainted with the Windows kernel synchronisation primitives (that is: mutex, event and semaphore), because I don't have the space talk too much about them. An excellent article on this topic was published way back in Issue 17 (*Sharing Data With The Win32 API* by John Chaytor). Another excellent source of information is *Programming Applications for Windows* (formerly called *Advanced Windows*) by Jeffrey Richter, which is a truly great book and a 'must read' for anybody digging into Windows API.

A Token

The first tool I'd like to present is a simple 'unique token' generator. It is not a synchronisation tool *per se*, but I still feel that it belongs to the synchronisation toolkit.

Sometimes when writing Windows applications we get a problem of unique identification. We have an entity (say a process, thread or object) and we need a way to uniquely identify it. The identification should be represented as a string that can be easily stored away or sent from one process to another. We also want to be able to tell if the entity (for which we have an identification) is still alive. That test should work correctly even if the process owning that entity crashed and went to computer heaven without executing any cleanup code *[Delphi apps always go to computer heaven when they crash, unlike C++ apps of course... Ed.]*.

These requirements arise from a class of client-server applications where client doesn't only want to know if the server is alive (this can easily be achieved with a simple mutex) but also if it is still the same server as in the beginning of the session.

Basically, we have to solve two problems: how to generate system-wide unique names and how to create a system-wide object that disappears if the owner dies. Both problems are simply solved by the use of the Win32 API.

To generate a unique name, we simply generate a GUID (a Globally Unique IDentifier). Microsoft guarantees that GUIDs are unique inside the system and they are even statistically unique worldwide, if the computer has a network card installed. Let that suffice. If you want to know more on GUIDs, fire up the *Collection 2002 CD* or open that huge box with your old copies of *The Delphi Magazine* and search for Issue 28 (*Delphi Meets COM: Part 1*) where Dave Jewell explained it all.

To generate a GUID we only have to call one API function, `CoCreateGuid` (declared in ActiveX.pas). It takes a parameter of type `TGUID`, which is filled with the generated GUID. To convert this value into a readable representation, we call another function, `GUIDToString` (declared in ComObj.pas). The result is a string which looks like this

```
{F91E84C9-7CE9-4657-ADBF-
  B5DE014F822C}
```

from which we will strip the leading and trailing curly brackets. The token creation code in the `TGpToken` constructor then tries to further reduce any possible conflicts by prefixing this result with a constant:

```
Gp/GpToken/04DC5A63-86AA-4439-
  8E9D-BC75C276968E/
```

Why? Because it is a good idea to use a hierarchical namespace for all system-wide identifiers you're using. For example, if you will be using this token name as a global (system-wide) mutex name, then you should make sure that no other process (possibly written by another programmer) would use the same name. Because of that, calling a global object `MyMutex`, for example, is not a very good thing. It is best to use a hierarchical form that starts with (say) the company name or the programmer's initials, followed by the name of the project, then maybe the name of the part of the project, and then

➤ *Listing 1: Creating a token.*

```
constructor TGpToken.Create;
var
  guid: TGUID;
begin
  CoCreateGuid(guid);
  gtTokenName := GUIDToString(guid);
  gtTokenName := CGpTokenPrefix + Copy(gtTokenName, 2, Length(gtTokenName)-2);
  Publish;
end; { TGpToken.Create }
```

```
procedure TGpToken.Publish;
begin
  if IsPublished then
    raise EGpSync.CreateFmt(sAlreadyPublished, [Token]);
  gtToken := CreateMutex(nil, false, PChar(Token));
  if gtToken = 0 then
    RaiseLastWin32Error
  else if GetLastError = ERROR_ALREADY_EXISTS then
    raise EGpSync.CreateFmt(sTokenAlreadyExists, [Token]);
end; { TGpToken.Publish }
```

➤ *Listing 2: Publishing a token.*

```
procedure TGpToken.Revoke;
begin
  if not IsPublished then
    raise EGpSync.CreateFmt(sNotPublished, [Token]);
  if not CloseHandle(gtToken) then
    RaiseLastWin32Error
  else
    gtToken := 0;
end; { TGpToken.Revoke }
```

➤ *Listing 3: Revoking a token.*

a GUID, generated just for that purpose.

Having said all that,actually I must admit that there is not much need to use the prefix when generating a token name. Remember that a GUID is used for the token name and that is pretty unique by itself. Still, I think you will agree that it doesn't hurt to get into good programming habits.

To recap, the code that creates a unique name and publishes it (I'll explain the 'publishing' part in a second) is shown in Listing 1.

Now that we have a unique name, we must make the name known to all the other processes sharing the same computer that this name is in use. The easiest way to achieve this is to create a mutex with that name. This is done in the `Publish` method, shown in Listing 2. There is really only one important line in the `Publish` method:

```
gtToken := CreateMutex(nil,
  false, PChar(Token))
```

All the rest is error handling. We want to make sure that the token is not already published, that the mutex was successfully created, and that no other mutex with the same name exists. Similarly, the method `Revoke` (Listing 3) un-publishes the token by closing the mutex handle. The job is done by the `CloseHandle(gtToken)` call, wrapped in some error handling.

I separated publishing and revoking the token into public subroutines because somebody might want to temporarily make some service (identified by the token) invisible. To achieve that, the programmer would only have to `Revoke` the token and `Publish` it back when the service was available again. This neatly corresponds to the Windows service model: create the `TGpToken` when the service is started, `Revoke` it when it is paused, `Publish` it when it is continued and destroy the token when the service is stopped.

To check if the token is published we will simply create the mutex (just like in the `Publish` method) and check the returned handle and error code. If the error code is 0, this means the mutex was successfully created, implying that no token with that name is published at the moment. If the error code is `ERROR_ALREADY_EXISTS`, a mutex with that name already exists. As you have surely already guessed, that means that the token is published. At the end we must close the mutex used for testing.

Testing could be implemented as a global function. I have decided, however, not to pollute the global namespace and therefore have moved this function into the `TGpToken` class. Of course, I'm not expecting fellow programmers to create an instance of this class every time they want to check if some token is published, so I made the checker a class function (see Listing 4). The minor disadvantage of this approach is that a programmer must always use the long form, `TGpToken.IsTokenPublished`, to call it.

The last part of the `TGpToken` class is just a small helper function (again created as a class function: see Listing 5) that creates a `TGpToken` object, returns its token and destroys it. It is useful if you only need the string representation of the token and not the publishing/revoking mechanism.

So how does this approach fare when the program auto-destructs without warning (possibly

➤ *Listing 4: Checking if a token is published.*

```
class function TGpToken.IsTokenPublished(token: string): boolean;
var
  testToken: THandle{CreateMutex};
begin
  testToken := CreateMutex(nil, false, PChar(token));
  if testToken = 0 then
    raise EGpSync.CreateFmt(sInvalidToken, [token])
  else begin
    try
      Result := (GetLastError = ERROR_ALREADY_EXISTS);
    finally
      if not CloseHandle(testToken) then
        RaiseLastWin32Error;
    end;
  end;
end; { TGpToken.IsTokenPublished }
```

➤ *Listing 5: Quick token generation.*

```
class function TGpToken.GenerateToken: string;
begin
  with TGpToken.Create do begin
    Result := Token;
    Free;
  end; //with
end; { TGpToken.GenerateToken }
```

because of an internal error, or because somebody kills it with a TerminateProcess)? It fares very well, thanks. Windows guarantees that any mutex owned by the process will be released and closed when that process terminates, even if it is killed with TerminateProcess. If the process dies, the mutex is closed, the token is no longer published, and everything behaves as expected.

For those of you that believe in using components at every step, I have created a component wrapper named TGpTokenComp (in the unit GpSyncComp). It publishes the properties Token (for read-only access to the token name) and IsPublished (a read-write property: toggling it causes the token to be published or revoked). Other TGpToken methods are available through the public methods.

In the GpSync unit (that is the one containing all synchronisation tools from this article) you will also find a TGpSWMRList class, which is a descendant of TObjectList with additional type checking. Nothing fancy, just a small helper that can make your code cleaner.

A Group

Enough of tokens, let's move on. The next item in my toolkit is a group. This is an encapsulation of a pool of entities (again, it can be a process, a thread, an object, or practically anything). An entity may join the group and leave it. Many entities (an unlimited number) can be members at the same time. The group may also be empty. We cannot tell how many members of the group there are, we can only tell whether the group is empty or not.

I have described a similar primitive, a file-based group, in my article on file system-based synchronization primitives (*Let's Cooperate*, in Issue 68). To keep things simple, and programming more portable, the group from this article implements the same interface as the file-based group.

The basic interface is simple: an entity can Join a group and Leave it. On Join, it will receive a flag indicating whether it was the first one to enter. On Leave, a flag will be set if the entity was last to leave the group.

There are two versions of Join and Leave, one with a simplified parameter list. This simpler version can be used when you don't need the 'is first/was last' status. There is also IsEmpty, which checks whether the group is empty but does not enter it. And there is a fairly trivial IsMember, which tells us if we have already joined the group.

As it is a system-wide object used for inter-process communication, every group must be named. To make the name as unique as possible (to prevent clashes with programs written by fellow programmers), you should make sure that the name is not easy to guess: possibly by using some hierarchical form as I have already described.

The group is implemented using two mutexes. One is used to represent the group (we'll call it the 'membership' mutex), the second prevents internal operations from being executed at the same time (the 'synchronisation' mutex). The names of both the mutexes are derived from the the name of the group. The first has the string MTXMember attached to the group name and the second is formed by adding MTXSync to the group name.

In other words, if the group is named Gp/Groups/MyGroup, then the membership mutex is named Gp/Groups/MyGroupMTXMember and the synchronisation mutex is Gp/Groups/MyGroupMTXSync.

A group functions in a similar way to the token: its presence is indicated by the state of the associated membership mutex. If the mutex exists, at least one process is in the group. When all the processes leave the group (that is, close the mutex), the mutex disappears. Although the general idea is simple, the implementation is slightly complicated by the fact that we want to be notified when the first process enters the group and the last process leaves it.

To join the group, the process calls the Join method (Listing 6). It first checks if the process has already joined the group (to handle nested Join/Leave calls)

```
procedure TGpGroup.Join(var isFirstMember: boolean);
begin
  if not IsMember then begin
    WaitForSingleObject(grSyncMutex, INFINITE);
    try
      grMemberMutex := CreateMutex(nil, false, PChar(MemberMutexName));
      if grMemberMutex = 0 then
        RaiseLastWin32Error
      else begin
        isFirstMember := (GetLastError = 0);
      end;
    finally ReleaseMutex(grSyncMutex); end;
  end;
  Inc(grTimesMember);
end; { TGpGroup.Join }
```

➤ *Listing 6: Joining a group.*

```
procedure TGpGroup.Leave(var wasLastMember: boolean);
begin
  if not IsMember then
    raise EGpSync.CreateFmt(sNotJoined, [Name]);
  Dec(grTimesMember);
  if grTimesMember = 0 then begin
    WaitForSingleObject(grSyncMutex, INFINITE);
    try
      CloseHandle(grMemberMutex);
      grMemberMutex := CreateMutex(nil, false, PChar(MemberMutexName));
      if grMemberMutex = 0 then
        RaiseLastWin32Error
      else begin
        wasLastMember := (GetLastError = 0);
        CloseHandle(grMemberMutex);
        grMemberMutex := 0;
      end;
    finally ReleaseMutex(grSyncMutex); end;
  end;
end; { TGpGroup.Leave }
```

➤ *Listing 7: Leaving a group.*

and then enters the critical section, governed by the synchronisation mutex (the reasons for that will become clear later, when we examine the Leave method). Then it creates the membership mutex and checks whether this was the first instance of the mutex (giving error code 0) or not (giving error code ERROR_ALREADY_EXISTS). At the end, the synchronisation mutex is released.

Leaving the group (see Listing 7) is more complicated. If it were not for the fact that we want to know the status of the 'was last' status, a simple CloseHandle(membershipmutex) call would suffice. To get the status of the group, we must then recreate the mutex, check the error status (as in the Join method) and close the mutex.

Just as in Join, Leave is also wrapped into a critical section. Without that precaution, another process could Join in the middle of the CloseHandle.. CreateMutex.. CloseHandle sequence, giving us a really weird result.

The last important method of the TGpGroup class is the function IsEmpty, which checks whether a group contains any members. Its implementation (shown in Listing 8) is similar to the Join method. IsEmpty first creates the mutex, then checks the error code, and closes the mutex. Of course, all that is only executed if this TGpGroup instance is not already a member of the group. In that case, the method knows that the group is not empty without any further checking.

If a process that is a member of a TGpGroup forcibly dies, it is automatically removed from the group, as all of its mutexes are automatically released and closed by Windows.

TGpGroup is accompanied by the matching component TGpGroupComp. It implements membership and emptiness checking through the properties IsMember and IsEmpty. The former can also be used to join or leave the group (by setting it to true or false, respectively). This method of joining or leaving the group, however, does not return the 'is first/was last' information. If you need that information, use the public Join and Leave methods.

To round things up, TGpGroup is also accompanied by the TObjectList descendant TGpGroupList.

A Counted Group

A logical extension of the simple group described above is a group that knows exactly how many members are actually present at the moment. In addition to that, it would be nice if we could limit the maximum number of group members.

If you are familiar with the Windows synchronisation primitives, you may have noticed that this is very similar to the way a *semaphore* operates. In fact, that is exactly how the counted group is implemented: a semaphore is used for counting and a mutex protects internal operations, just like in the TGpGroup class.

The use of a semaphore also brings up some problems that I was not aware of when coding the counted group. Only when writing this text did I find an excellent article *Sharing Data With The Win32 API* (by John Chaytor, in Issue 17), where the author stated: *Windows does not clean up a Semaphore object correctly if the application terminated without calling ReleaseSemaphore.* That was so weird that I couldn't believe it without checking it out. Of course, Mr Chaytor was right: semaphores on Win32 are unsafe. It is possible for a process to die without releasing a semaphore that it owns.

To put you at ease, let me say that all is not as black as it seems (or nobody would be using semaphores at all). It is hardly possible that this will ever happen to a well-written process under normal circumstances (by well-written I mean a program that is protecting its resources and matches every acquisition of a semaphore with a call to ReleaseSemaphore). But even a well-written program can die while it has a semaphore acquired. Somebody may kill it from the Task Manager's Processes page or with another utility that terminates it with a call to TerminateProcess. Or somebody (maybe even the program itself) can kill the thread that is holding the semaphore with a call to TerminateThread. But the most probable cause of problems is the programmer him/herself. In the heat of a debugging session it is very easy to reset the execution (Ctrl-F2) when the semaphore is acquired, and that would of course cause it not to be released.

Sadly, there is no easy way to recover from such a situation. It may help to log off and re-login, then again it may not (if the semaphore in question was owned by a service) and only a reboot would help.

The moral of the story is that if you are using semaphores (or counted groups, which are based on semaphores), be very, very careful and always expect the unexpected.

So now to the implementation. As I have already said, a counted group uses one mutex to protect internal operations and one semaphore to keep a count of free slots in the group. Both are created and initialised in the constructor, the semaphore is created with an initial count and maximum count

➤ *Listing 8: Checking whether the group is empty.*

```
function TGpGroup.IsEmpty: boolean;
begin
  Result := false;
  if not IsMember then begin
    WaitForSingleObject(grSyncMutex, INFINITE);
    try
      grMemberMutex := CreateMutex(nil, false, PChar(MemberMutexName));
      if grMemberMutex = 0 then
        RaiseLastWin32Error
      else begin
        Result := (GetLastError = 0);
        CloseHandle(grMemberMutex);
        grMemberMutex := 0;
      end;
    finally ReleaseMutex(grSyncMutex); end;
  end;
end; { TGpGroup.IsEmpty }
```

of MemberLimit (a parameter of the constructor representing the maximum number of group members).

To join the counted group, the process calls the Join method. It takes one or two parameters, just like the TGpGroup.Join.

Joining the counted group is slightly tricky. To check if the caller is the first member of the group, we must examine the semaphore count. But the only way to get it is to wait on the semaphore for the second time and then release it with a call to ReleaseSemaphore, which can return the previous semaphore count (just as it was before the call to ReleaseSemaphore).

All three semaphore operations (wait, wait again and release) must be protected with the mutex (or the result would be hopelessly unpredictable). So the main question is: when should we acquire this mutex? Just after the semaphore is acquired for the first time? No, that would leave a short time when operations are not protected. How about just before the semaphore is acquired? This is better, but if the semaphore could not be acquired because the group is full, we would have to release the mutex (to allow another process to leave the group) and try again after some time. In effect, we would have to reproduce the test-and-wait loop used in the file system-based group I described in Issue 68.

Luckily, Windows offers us a better alternative: we can wait on both the semaphore and the mutex at the same time. The kernel will make sure that we either acquire both or neither. We only have to call WaitForMultipleObjects and set the bWaitAll parameter to True. After that, the Join code (shown in Listing 9) is simple. It first increments the internal variable which counts how many times this object has joined the group (TGpCounterGroup supports nested Join/Leave calls from the same object). Then it checks whether this is the first member of the group, by retrieving the current number of members. If this is equal to the MemberLimit-1, this is the first member (this object has already joined the group and has decremented the semaphore count). At the end, the code releases the mutex, allowing other internal operations to execute.

The internal function UnprotectedNumMembers (see Listing 10), which is also called from the public function NumMembers, was mostly described two paragraphs above. It waits on the semaphore and then releases it, retrieving the semaphore count along the way. If the semaphore cannot be acquired, the group is already full and the function returns the group's MemberLimit parameter.

Leaving the group is much simpler. First we decrement the internal membership counter, and then

➤ *Listing 9: Joining a counted group.*

```
function TGpCountedGroup.Join(timeout: DWORD; var isFirstMember: boolean):
  boolean;
var
  handles: array [0..1] of THandle;
  waitRes: DWORD;
begin
  handles[0] := cgrSemaphore;
  handles[1] := cgrMutex;
  waitRes := WaitForMultipleObjects(2,@handles,true,timeout);
  if (waitRes <> WAIT_OBJECT_0) and (waitRes <> (WAIT_OBJECT_0+1)) then
    Result := false
  else begin
    try
      Inc(cgrTimesMember);
      isFirstMember := (UnprotectedNumMembers = (MemberLimit-1));
    finally
      if not ReleaseMutex(cgrMutex) then
        RaiseLastWin32Error; { this really shouldn't happen }
    end;
    Result := true;
  end;
end; { TGpCountedGroup.Join }
```

➤ *Listing 10: Retrieving a group member count.*

```
function TGpCountedGroup.UnprotectedNumMembers: integer;
var
  memberCount: integer;
begin
  if WaitForSingleObject(cgrSemaphore,0) <> WAIT_OBJECT_0 then
    Result := MemberLimit
  else begin
    if not ReleaseSemaphore(cgrSemaphore,1,@memberCount) then
      RaiseLastWin32Error; { this really shouldn't happen }
    Result := (MemberLimit - (memberCount+1));
  end;
end; { TGpCountedGroup.UnprotectedNumMembers }
```

➤ *Listing 11: Interface of the Single Writer Multipler Readers guard.*

```
TGpSWMR = class
  constructor Create(swmrName: string);
  destructor  Destroy; override;
  procedure Done; virtual;
  function  WaitToRead(timeout: DWORD): boolean; virtual;  // true if allowed
  function  WaitToWrite(timeout: DWORD): boolean; virtual; // true if allowed
  property Access: TGpSWMRAccess read gwrAccess;
  property Name: string read gwrName;
end; { TGpSWMR }
```

➤ *Listing 12: Getting the read access.*

```
function TGpSWMR.WaitToRead(timeout: DWORD): boolean;
var
  prevCount: longint;
begin
  if WaitForSingleObject(gwrMutexNoWriter, timeout) <> WAIT_TIMEOUT then begin
    ReleaseSemaphore(gwrSemNumReaders, 1, @prevCount);
    if prevCount = 0 then
      ResetEvent(gwrEventNoReaders);
    ReleaseMutex(gwrMutexNoWriter);
    Result := true;
  end else
    Result := false;
end; { TGpSWMR.WaitToRead }
```

we leave the group by calling the `ReleaseSemaphore` routine. This is protected with the mutex to prevent it from interfering with another object executing the `Join` method.

The other counted group methods are also quite simple and I won't describe them here. Examine the source code and I'm sure you will understand how they work easily enough.

The matching component `TGpCountedGroupComp` exposes the counted group functionality in the same manner as `TGpGroupComp`. Of course, `TGpCountedGroup` is also accompanied by the `TObjectList` descendant `TGpCountedGroupList`.

```
function TGpSWMR.WaitToWrite(timeout: DWORD): boolean;
var
  handles: array [1..2] of THandle;
begin
  handles[1] := gwrMutexNoWriter;
  handles[2] := gwrEventNoReaders;
  Result := (WaitForMultipleObjects(2, @handles, true, timeout) <> WAIT_TIMEOUT);
end; { TGpSWMR.WaitToWrite }
```

➤ *Listing 13: Getting the write access.*

```
procedure TGpSWMR.DoneReading;
var
  handles: array [1..2] of THandle;
begin
  handles[1] := gwrMutexNoWriter;
  handles[2] := gwrSemNumReaders;
  WaitForMultipleObjects(2, @handles, true, INFINITE);
  if WaitForSingleObject(gwrSemNumReaders, 0) = WAIT_TIMEOUT then
    SetEvent(gwrEventNoReaders) // last reader
  else
    ReleaseSemaphore(gwrSemNumReaders, 1, nil);
  ReleaseMutex(gwrMutexNoWriter);
end; { TGpSWMR.DoneReading }
```

➤ *Listing 14: Releasing the read access.*

Writers And Readers

The last tool in my toolkit, for now, is a good old Single Writer Multiple Readers guard (or SWMR for short), with a few twists. The SWMR protects a resource that can be accessed either by a single writer or by multiple readers. It can be found in different Delphi libraries, and even in Delphi itself: the `SysUtils` unit has the `TMultiReadExclusiveWriteSynchronizer` class that implements an inter-process SWMR guard.

The `TGpSWMR` interface (shown in Listing 11) is quite simple. The caller uses `WaitToRead` in order to get read access to the protected resource and `WaitToWrite` to get write access to the resource. In both cases, access is released with the `Done` method. In addition, multiple `WaitTo`/`Done` calls can be nested. For example, a process can call `WaitToRead` twice in a row without an intermittent `Done` call. After that, the process must call `Done` twice in order to release the protected resource.

It is also possible to call `WaitToRead` when a process already owns a write lock obtained by `WaitToWrite`. In that case, the process already has an exclusive access to the protected resource, and `WaitToRead` call does not have to do anything. Of course, the process must still call `Done` twice to release the resource.

Looking at it the other way around: calling `WaitToWrite` when the process already owns a read lock is not permitted and will raise an exception.

The basic `TGpSWMR` implementation is directly copied from Jeffrey Richter's *Advanced Windows*. On top of that that existing framework I have added some simple code that implements nested `WaitTo` calls.

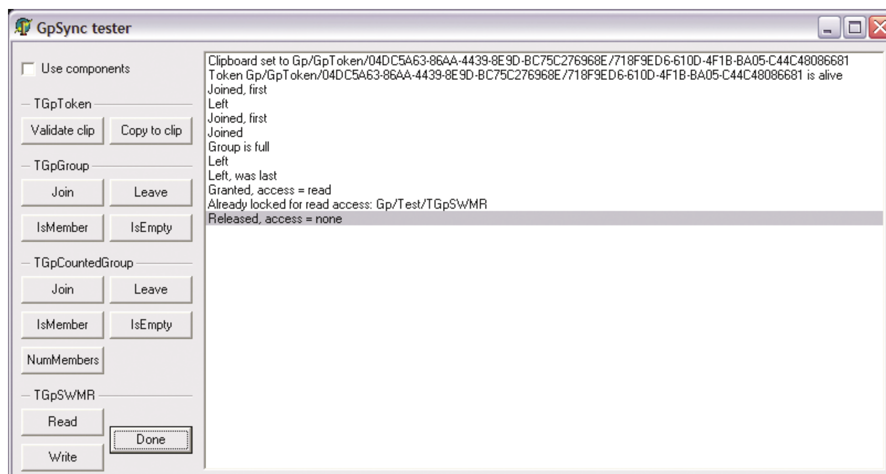
```
procedure TGpSWMR.DoneWriting;
begin
  ReleaseMutex(gwrMutexNoWriter);
end; { TGpSWMR.DoneWriting }
```

➤ *Listing 15: Releasing the write access.*

Mr Richter's code uses three primitives to implement the SWMR guard: a mutex to indicate that there is no writer, a semaphore to count active readers, and lastly an event to signal when no readers are accessing the protected resource. A semaphore is created with an initial count of 0, to indicate that no readers are currently accessing the resource.

Listing 12 shows the important parts of the `WaitToRead` method (all of the code which handles the nested `WaitTo`/`Done` calls has been removed for clarity: it's on the disk of course). It read-locks the resource by waiting on the mutex `gwrMutexNoWriter`. That wait is only successful if there is no writer and the mutex is not acquired. When the mutex is acquired, the code releases the semaphore `gwrSemNumReaders` in order to increment the number of active readers. The `ReleaseSemaphore` call returns the previous semaphore count: if this previous count is zero, we are the

➤ *Figure 1*



first reader and we have to reset the event `gwrEventNoReaders`, indicating that at least one reader exists. At the end, the code releases the `gwrMutexNoWriter` mutex.

The `WaitToWrite` routine (for which simplified code is shown in Listing 13) is simpler: it only waits on both `gwrMutexNoWriter` and `gwrEventNoReader` to become signalled at the same time. This can only happen when there is no writer and no reader, which is exactly when we can write-lock the protected resource.

When read access is released, the public method `Done` calls the private workhorse routine `DoneReading` (shown in Listing 14). It waits on `gwrMutexNoWriter` and `gwrSemNumReader` at the same time. The former must be acquired in order to prevent a race condition, while the latter is waited on simply to decrease the semaphore count (indicating we have one less reader). The code then tries to wait on the semaphore again to check if this is the last reader. If this is the case, the event `gwrEventNoReader` is signalled, otherwise the semaphore count is returned back to the previous value. At the end, `gwrMutexNoWriter` is released.

Again, the code to release the write access is simpler (shown in Listing 15). It only releases the mutex `gwrMutexNoWriter` (acquired in the `WaitToWrite` method).

Because it uses a semaphore, `TGpSWMR` doesn't really like its owner to be killed. Handle it with care!

Needless to say, `TGpSWMR` is accompanied by the component `TGpSWMRComp` and the `TObjectList` descendant `TGpSWMRList`.

Conclusion

I hope you will find the techniques and code presented in this article useful in your projects.

Also included on the companion disk is a demonstration program (shown in Figure 1) that will help you understand and test all of the tools from this article.

That is all for now. In the next article in this occasional series I will create a better shared memory object. Better than what? All other shared memory objects floating around the net? Yes, of course!

Primoz Gabrijelcic is the R&D Manager of FAB d.o.o. in Slovenia. You can contact him by email at gp@fab-online.com

All code in this article is freeware and may be freely reused in your own applications.

To advertise in The Delphi Magazine call Leigh Foster on +44 (0)1234 241454 or email her at leighf@itecuk.com

The Best N-Tier Development System is now Better!

ASTA 3

Potent New Features - ***PLUS** a **700 page PDF Manual*** - make it easier than ever to build secure, Thin-client, Cross Platform Data Base applications.

Avoid burning development time on low-level issues! Take a moment and take the ASTA Tour.

ASTA Technology Group, Inc.
www.astatech.com
www.astawireless.com
info@astatech.com

TDMWeb

Your website in safe hands www.TDMWeb.com

Want More From The Web?

Is your website a hungry, snarling, beast, which eats up server resources like they're going out of fashion? Does your web host keep suggesting you find somewhere else to go?

TDMWeb can help.

Whether you want a custom hosting package with extra storage space, extra data transfer, or even a dedicated server all to yourself (totally managed by us, hassle-free for you), or someone to design your site, or someone to specify and write your web applications, **we'll sort it**.

And we'll give you very competitive prices, as well as our renowned customer service.

Call +44 (0)870 740 760 or email us at sales@tdmweb.com to find out more