Let's Cooperate

File system-based synchronisation primitives

by Primoz Gabrijelcic

Every serious Windows programmer who has developed at least one multi-threaded application knows about thread synchronisation. You know what I mean: critical sections, mutexes, semaphores, signals and so on. Those *synchronisation primitives* are important tools for building complex, multi-threaded applications. But to write truly distributed applications, running on different computers, we need new tools. Standard Windows synchronisation primitives only work within one computer: they won't help us when synchronising processes on many computers, which are most probably running different operating systems too.

Obviously we need a new set of tools, synchronisation primitives for a network environment, but what should we base them on? Sockets (TCP or UDP) are a nice idea, but there is probably a simpler solution: the file system. Yes, we can use good old methods like locked files and with the help of a decent file server breathe new life into these techniques. (I'll talk more about this later, but for the curious reader I'll mention that Windows NT is 'decent' enough in our context and even Windows 9x would do the job.)

Before we move on, remember one thing: file-based synchronisation is slow. Extremely slow. Your application could calculate a few hundred thousand prime numbers while it is waiting on a mutex to be acquired. That slow. So please try to remember that, and use these primitives as little as possible.

If you don't know that much about classical synchronisation primitives, you can read more about them in *The Delphi Magazine: Happiness Is An Option* (Issue 55) and *Sharing Data With The Win32 API* (Issue 17). I would also recommend the book *Advanced Windows* by Jeffrey Richter, which covers many other important Windows topics too. Or maybe *Operating System Principles* by Per Brinch Hansen, if you can get your hands on it: it's an oldie, but a goldie too. As we'll also implement a message-passing class, you may want to check up on on standard Win32 mechanisms in this area, too. Recommended reading is again Richter's book or the *Win32 Inter-Process Communication* article from Issue 50.

A Decent File Server

I already mentioned that we need a decent file server. Luckily, almost anything you will encounter will be good enough. For example, the file server has to allow us to work with exclusive access or in shared-read mode (ie, many processes can read the file at the same time). OK, that is not a problem. More important is that the file server should release an exclusively opened file after some time if our process (the one that was keeping the file open) dies. These requirements are not too onerous and any network operating system (eg Windows NT, Netware and Unix) should satisfy them. I say 'should' as this code was only tested on Windows NT and 2000. If you will be using other platforms, test long and test well before you depend on my code. However, I'm working on porting this code to Linux, using Kylix of course, and will report back on that project soon!

Let the coding begin. Standard texts on synchronisation primitives almost exclusively start with critical sections. There is a good reason for that: critical sections in the Win32 environment work only within one process and, because of that, they are simple to explain and use. As we will focus on the networked environment, single process solutions are not important to us and we'll start with a file-based mutex. After that, we'll develop a file-based critical section (a simplification of a file-based mutex), a file-based group (a non-standard but very useful primitive), a file-based event and a file-based message. Finally, I'll discuss some advanced topics: semaphores and single writer multiple reader guards.

A File-Based Mutex

We'll derive all our file-based synchronisation classes from the parent class `TGpFileSynchroObject` (Listing 1). Besides some functions which are used in all the derived classes, it exposes two public properties: the name of the file used for synchronisation purposes and the retry delay. We need the latter because all our file-based synchronisation objects will periodically check if some condition is satisfied. Between checks, they will sleep for `RetryDelay` milliseconds (100 by default).

And now to our mutex. Basically, a mutex is a token that can only be owned by one process at a time. To work on critical (shared) data, a process takes the token (*acquires the mutex*) and when it is finished, it returns the token (*releases the mutex*). For safety reasons, all owned mutexes must automatically be released when the process

➤ *Listing 1: Ancestor of all file synchronisation objects.*

```
TGpFileSynchroObject = class
private
  fsoFileName  : string;
  fsoRetryDelay: integer;
protected
  constructor Create(syncFile: string; alwaysCheckForWriteAcc: boolean);
  procedure CheckForWriteAccess(folder: string);
  function  Elapsed(start: int64; timeout: DWORD): boolean;
public
  property  RetryDelay: integer read fsoRetryDelay write fsoRetryDelay;
  property  SyncFile: string read fsoFileName;
end; { TGpFileSynchroObject }
```

terminates (whether normally or abnormally) or if the computer that is running the process crashes or is disconnected from network (and with that from other processes). As you may have expected from my introduction, a simple file satisfies all these needs. If we open it in read-only deny-all mode, we have acquired the mutex. When we close it, the mutex is released. Only one process at a time can own the mutex (file) because of the deny-all mode (that tells the file server nobody else may open the file while we're working on it).

Because we'll be opening the mutex in read-only mode, we can make the mutex file read-only or we can even protect the folder with the mutex file in so that no ordinary user can write into it. Of course, our file-based mutex object cannot create a mutex file in that case. Some other program (maybe a configuration or installation program running with administrative rights) must create this file in advance. The file-based mutex therefore allows us to design clean solutions where clients only require read access to the shared folder and only one program, used by the administrator, requires write access to this folder.

Let's take a quick look at a class to encapsulate our file-based mutex: `TGpFileMutex` (Listing 2). The constructor stores the mutex file name in a local variable and sets the 'delete on release' flag. Setting this flag to `true` specifies that the mutex file must be deleted when released. Of course, the process owning the mutex must have enough rights to the file and folder.

The most important method in the `TGpFileMutex` class is `Acquire`. It tries to acquire the mutex for at most `timeout` milliseconds and returns `true` if the mutex was successfully acquired, `false` if not, and `EGpFileSync` exception on illegal use (which implies programming error). `Timeout` can be set to `0` for no waiting or `INFINITE` (declared in Windows.pas) for endless waiting.

Other methods are simpler. `Acquired` checks if the mutex is already acquired and `Release` releases it (or, if the mutex is not acquired, raises the `EGpFileSync` exception). The destructor checks if the mutex is acquired and, if so, releases it.

```
TGpFileMutex = class(TGpFileSynchroObject)
private
  fmDelete: boolean;
  fmHandle: THandle;
public
  constructor Create(syncFile: string; deleteOnRelease: boolean = false);
reintroduce;
  destructor  Destroy; override;
  function  Acquire(timeout: DWORD): boolean;
  function  Acquired: boolean;
  function  IsFree(timeout: DWORD): boolean;
  procedure Release;
end; { TGpFileMutex }
```

➤ *Listing 2: File-based mutex.*

```
function TGpFileMutex.Acquire(timeout: DWORD): boolean;
var
  flag : DWORD;
  err  : DWORD;
  start: int64;
begin
  if Acquired then
    raise EGpFileSync.CreateFmt(SAlreadyAcquired,[SyncFile])
  else begin
    flag := FILE_ATTRIBUTE_NORMAL;
    if fmDelete then
      flag := flag OR FILE_FLAG_DELETE_ON_CLOSE;
    start := GetTickCount;
    repeat
      fmHandle := CreateFile(PChar(SyncFile),GENERIC_READ,0,nil,OPEN_ALWAYS,
        flag,0);
      if fmHandle = INVALID_HANDLE_VALUE then begin
        err := GetLastError;
        if err in FILE_SHARING_ERRORS then
          Sleep(RetryDelay)
        else
          raise EGpFileSync.CreateFmt(SCannotAccessFile,
            [SyncFile,SysErrorMessage(err)]);
      end else
        err := 0;
    until (err = 0) or Elapsed(start,timeout);
    Result := (err = 0);
  end;
end; { TGpFileMutex.Acquire }
```

➤ *Listing 3: Acquiring a mutex.*

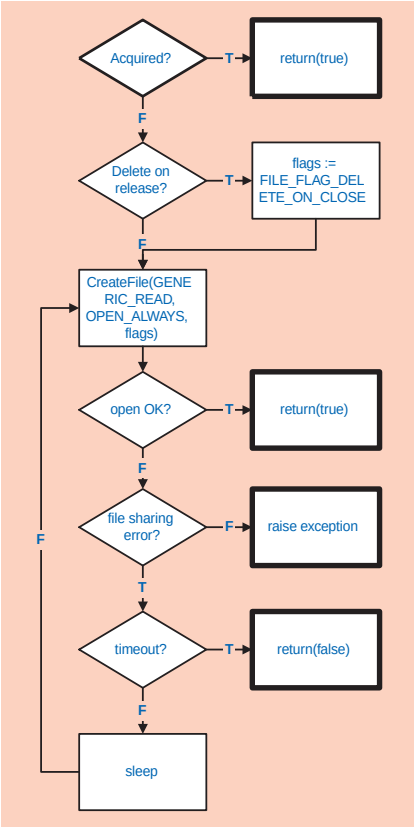
The `IsFree` method is a helper function that checks whether the mutex is available. It does that by acquiring and immediately releasing it. If the mutex cannot be acquired it is not available. We'll use this helper later, when we implement the single writer multiple reader guard and mutex monitoring components.

Mutex Explained

Let's now take a detailed look at how `TGpFileMutex.Acquire` operates (see the flowchart in Figure 1 and implementation in Listing 3).

First, check if the mutex is already acquired (by calling `Acquired`). If so, treat this as a programmer error and raise the `EGpFileSync` exception.

Next, check if the 'delete on release' flag is set. If so, open the



➤ *Figure 1: Acquiring a mutex.*

file with FILE_FLAG_DELETE_ON_CLOSE flag set. This tells Windows that the file should be deleted when closed.

Now try to open the file with GENERIC_READ (read-only) access, sharing mode 0 (deny all) and OPEN_ALWAYS creation disposition (open existing file or create new).

If CreateFile succeeded the mutex is acquired: leave the file open and return true. If CreateFile failed with something other than a sharing error (ERROR_SHARING_VIOLATION or ERROR_LOCK_VIOLATION), treat this as a programmer error (the file name is invalid, folder does not exist, or similar) and raise the EGpFileSync exception.

Next, check if the allotted time has elapsed. If so, return false and exit. Otherwise, sleep for RetryDelay milliseconds and try again to open the file, following the procedure through just as before.

Release is simpler: it checks if the mutex is acquired (raising the EGpFileSync exception if not) and closes the mutex file. Acquired just checks if the mutex file is open.

To use the mutex for synchronisation (for example, to access a shared resource), first Acquire it and at the end Release it. Always treat the file mutex as a critical resource and put all access to the shared resource inside a try..finally block. Listing 4 shows a simple demonstration of synchronisation with TGpFileMutex.

➤ *Listing 4: Synchronising access to a shared resource with TFileMutex.*

```
procedure MutexDemo;
var
  fileMutex: TGpFileMutex;
begin
  fileMutex := TGpFileMutex.Create('c:\test.lck');
  try
    if fileMutex.Acquire(1000) then
      try
        // work with shared resource
      finally fileMutex.Release; end
    else
      // failed to access shared resource
  except
    on E: EGpFileSync do
      // report error
  end;
end; { MutexDemo }
```

Critical Section

I already mentioned that critical sections as an inter-process mechanism don't really apply to our multi-computer environment. However, it can be quite helpful to have a class that looks and works just like Delphi's TCriticalSection. So I built a simple descendant of TGpFileMutex called TGpFileCriticalSection (Listing 5). Basically, it creates wrappers for the Acquire and Release functions and adds the aliases Enter and Leave. With an additional twist, you can nest Enter and Leave calls, just as in the equivalent TCriticalSection methods. This allows for an easier transition from existing code using TCriticalSection or Windows' critical sections.

➤ *Listing 5: File-based critical section.*

```
TGpFileCriticalSection = class(TGpFileMutex)
private
  nestCount: integer;
public
  procedure Acquire; reintroduce;
  procedure Enter;
  procedure Leave;
  procedure Release; reintroduce;
end; { TGpFileCriticalSection }
```

➤ *Listing 6: Implementation of TGpFileCriticalSection.Acquire and TGpFileCriticalSection.Release.*

```
procedure TGpFileCriticalSection.Acquire;
begin
  if nestCount = 0 then
    inherited Acquire(INFINITE);
  Inc(nestCount);
end; { TGpFileCriticalSection.Acquire }
procedure TGpFileCriticalSection.Release;
begin
  Dec(nestCount);
  if nestCount <= 0 then
    inherited Release;
end; { TGpFileCriticalSection.Release }
```

Let me explain this. TGpFileMutex takes care that you don't call Acquire twice in a row, like this:

```
fm1.Acquire(0);
fm1.Acquire(0);
fm1.Release;
fm1.Release
```

This may force us to think a little more about how we will acquire it, but makes it behave like standard Windows mutex. Critical sections in Windows behave differently: you can enter a critical section while you are already in it. In TGpFileCriticalSection terms, you can use a critical section in the following manner:

```
cs1.Enter;
cs1.Enter;
cs1.Leave;
cs1.Leave;
```

Of course, only the last cs1.Leave actually leaves the critical section.

To implement this, TGpCriticalSection uses a private nesting counter, which counts how many Enter calls are not yet matched with corresponding Leave calls. TGpCriticalSection.Acquire (or Enter, the alias for it) will only call inherited Acquire(INFINITE) if the nesting count is zero. Similarly, TGpCriticalSection.Release (or Leave) will only call inherited Release when nesting count drops to zero. Acquire and Release are shown in Listing 6.

TGpFileCriticalSection can be useful for quick and dirty programs and internal solutions. Just don't use it in real-world applications, as it doesn't offer an option to limit the duration of Acquire. A commercial application that locks indefinitely is a very bad idea!

Group: A Global Pool

So, should we start writing a file-based semaphore now? No! While the mutex and critical section are quite simple to implement using the file system, a semaphore is not. I'll discuss this later, but for now just believe me: writing a file-based semaphore is a tough job.

Rather than that, I'll implement something simpler and almost as useful: a group. I must warn you, a group as presented here is entirely my invention. I don't know if there is any well-established definition or implementation of a group, so I made my own.

In this article I'll define a group as a pool of processes. Processes may join or leave the group. There can be an unlimited number of processes in the group, or the group may be empty. We can't tell how many processes are in the group, only if it's empty or not.

`TGpFileGroup` (Listing 7) encapsulates that behaviour: the owner can `Join` the group and `Leave` it. On `Join`, `TGpFileGroup` will set a flag if we are the first process to enter the group. On `Leave`, the flag will be set if we are the last to leave the group. Of course, we'd like the owner to automatically leave the group if the program crashes.

```
TGpFileGroup = class(TGpFileSynchroObject)
private
  fgDelete: boolean;
  fgHandle: THandle;
  fgLock  : TGpFileMutex;
public
  constructor Create(syncFile: string; deleteOnRelease: boolean = false);
    reintroduce;
  destructor  Destroy; override;
  function  IsEmpty(timeout: DWORD; var emptyGroup: boolean): boolean;
  function  IsMember: boolean;
  function  Join(timeout: DWORD; var isFirstMember: boolean): boolean; overload;
  function  Join(timeout: DWORD): boolean; overload;
  function  Leave(timeout: DWORD; var wasLastMember: boolean): boolean; overload;
  function  Leave(timeout: DWORD): boolean; overload;
end; { TGpFileGroup }
```

➤ *Listing 7: File-based group.*

There are two versions of `Join` and `Leave`: one with a simplified parameter list. The simpler version could be used when we don't want to know if we were the first or last member of the group. `IsEmpty` checks if the group is empty but does not enter it. `IsMember` tells us if we have already `Join`ed the group.

The group uses two files, one for group and another for an associated mutex. It can work with read-only folders and files if both the group and mutex files are created in advance. To simplify the interface, only the name of the group file is required. The `TGpFileGroup` constructor will append `_lck` to it and use that for the mutex file name.

We will use file sharing to implement a group. Each process that is a member of the group will keep the group file open in read-only share-read mode (which allows other processes to open the group file in the same mode). To check whether the group is empty, we open the file in deny-all mode. If that fails, at least one process is already a member of the group. Because that requires opening and closing the file, which is clearly not an indivisible (atomic) operation, we'll use an associated mutex to create a critical section, which only one `TGpFileGroup` will be able to enter at a time.

The complete flowchart for `Join` is shown in Figure 2. First, check if we are already a member of this group. If so, treat this as a programmer error and raise the `EGpFileSync` exception.

Next, acquire the mutex, returning `false` on timeout. Now try to open the file in read-only deny-all create-if-not-exists mode.

If the file open failed because of a sharing error, set a flag indicating that we are not the first and go to the next step. If the open failed because of some other error, raise an exception. If the file opened without a problem, set a flag indicating that we are the first in the group and go to the next step.

Now close the file, then reopen it in read-only share-read mode. Finally, release the mutex.

`Leave` is similar to `Join` except that it works in reverse order: first it leaves the group then checks if the group is empty (by trying to re-join it without sharing). Figure 3 shows the `Leave` flowchart.

First, check if we are not a member of this group. If we are, treat this as a programmer error and raise the `EGpFileSync` exception. Next, acquire the mutex, returning `false` on timeout.

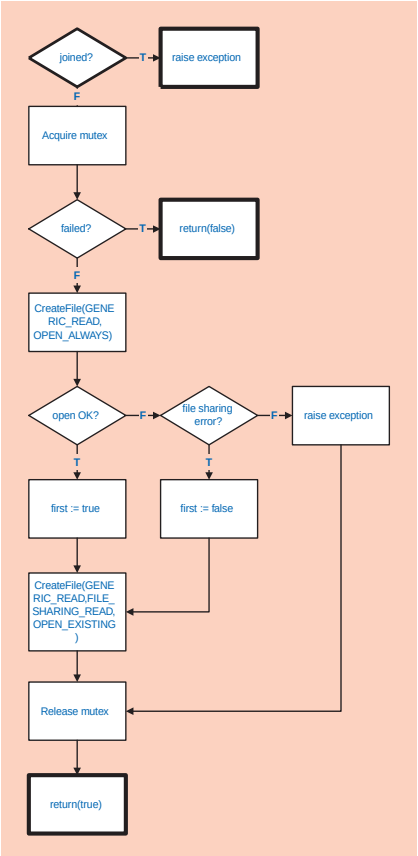
Now close the handle to the group file (the one that was opened and stored in `Join`), and try to open the file in read-only deny-all, do-not-create mode.

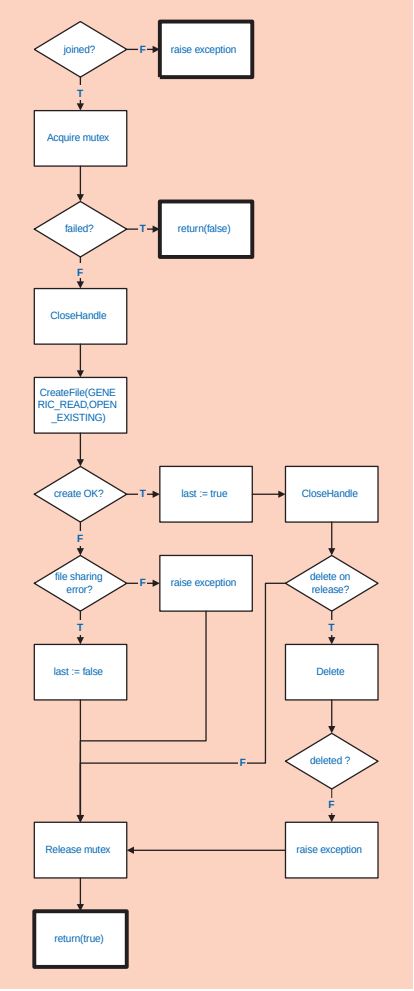
If the file opened without a problem, close it. If 'delete on release' is set, try to delete it. Set a flag indicating that we were the last in the group, then go to the next step. If the open failed because of a sharing error, set a flag indicating that we were not the last and go back to the step where we closed the file handle. If the open failed because of some other error, raise an exception.

Finally, release the mutex.

The implementation of `IsEmpty` is similar to `Join`, except that it closes the file as soon as it

➤ *Figure 2: Joining a group.*





➤ *Figure 3: Leaving a group.*

determines whether we are the first member of the group or not.

First, check if we are already a member of this group. If we are, set a flag indicating that the group is not empty. Next acquire the mutex, returning `false` on timeout.

Now try to open the file in read-only, deny-all, create-if-not-exists mode. If the open failed because of a sharing error, set a flag indicating that the group is not empty and go to the next step. If the open failed because of some other error, raise an exception. If the file opened without a problem, set a flag indicating that the group is empty, close the file, and go to the next step 4.

Finally, release the mutex.

A group is not a synchronisation tool *per se*. True, you can use it as one (`Join` the group, use the resource only if the group was empty before joining, `Leave` the group), but mostly you will want to use it as a flag indicating that some kind of activity is in progress. For example, you can set an application so that it joins the group when the app starts and leaves it when it exits; then you set up an administration utility for the same application so it will warn you if any client program is running. An example of such an application/administration pair is shown in Listings 8 and 9. This also demonstrates that a program leaves a group if it is improperly terminated. Run the application, kill it (with `Ctrl-C`), then run AdminUtil and you'll see how it correctly determines the group is empty.

➤ *Listing 8: TGpFileGroup demo, Application part.*

```
program FileGroupApplication;
{$APPTYPE CONSOLE}
uses
  SysUtils,GpFileSync;
var
  grp: TGpFileGroup;
begin
  grp := TGpFileGroup.Create('demo.grp');
  try
    if not grp.Join(5000) then
      Writeln('Cannot join.')
    else begin
      try
        Writeln('Application running, press Enter to terminate...');
        Readln;
      finally grp.Leave(5000) end;
    end;
  finally FreeAndNil(grp); end;
end.
```

➤ *Listing 9: TGpFileGroup demo, AdminUtil part.*

```
program FileGroupAdmin;
{$APPTYPE CONSOLE}
uses
  SysUtils, GpFileSync;
var
  grp: TGpFileGroup;
  grpEmpty: boolean;
begin
  grp := TGpFileGroup.Create('demo.grp');
  try
    if not grp.IsEmpty(5000,grpEmpty) then
      grpEmpty := false; // cannot check group status - assume not empty
    if not grpEmpty then
      Writeln('At least one Application is running, be careful!');
  finally FreeAndNil(grp); end;
end.
```

Ping!

We have one more traditional synchronisation primitive to cover: an event. It is not a resource allocation primitive, like a mutex and a critical section (and in some ways like a group), but a communication primitive. With an event one process can tell another that something has happened or that some condition has occurred. For example, a process can use an event to inform another process that a data acquisition phase has completed and it should begin processing the data. We say that the first process *signals* the event and the second one *waits for* it and *resets* it.

Basically, a file-based event is not much more than a file. The presence of a file represents a signalled event and absence of the file represents a reset, inactive, event. The `TGpFileEvent` interface (Listing 10) and implementation are almost trivial (especially compared to a mutex and a group) but some things are worth noting:

- `TGpFileEvent` is not based on a mutex and doesn't use a mutex.
- The synchronisation file is not deleted if the signalling process crashes. If this doesn't suit you, use `TGpFileMessage` from the next section.
- `TGpFileEvent` requires write access to the synchronisation folder, so the signalling process must be able to create a file there and the waiting process must be able to delete it.
- There is no 'delete on release' parameter. We already stated the reason: deleting the synchronisation file resets an event.

```
TGpFileEvent = class(TGpFileSynchroObject)
public
  constructor Create(syncFile: string); reintroduce;
  function Reset: boolean;
  function Signal: boolean;
  function WaitFor(timeout: DWORD; reset: boolean): boolean;
end; { TGpFileEvent }
```

➤ *Listing 10: File-based event.*

```
function TGpFileEvent.Signal: boolean;
var
  h  : THandle;
  err: DWORD;
begin
  h := CreateFile(PChar(SyncFile),GENERIC_READ,0,nil,CREATE_NEW,
    FILE_ATTRIBUTE_NORMAL,0);
  if h = INVALID_HANDLE_VALUE then begin
    err := GetLastError;
    if err = ERROR_FILE_EXISTS then
      Result := false
    else
      raise EGpFileSync.CreateFmt(SCannotCreateFile,
        [SyncFile,SysErrorMessage(err)]);
  end else begin
    CloseHandle(h);
    Result := true;
  end;
end; { TGpFileEvent.Signal }
function TGpFileEvent.Reset: boolean;
begin
  if not FileExists(SyncFile) then
    Reset := false
  else begin
    if Windows.DeleteFile(PChar(SyncFile)) then
      Result := true
    else
      raise EGpFileSync.CreateFmt(SCannotDeleteFile,
        [SyncFile,SysErrorMessage(GetLastError)]);
  end;
end; { TGpFileEvent.Reset }
```

➤ *Listing 11: Implementation of TGpFileEvent.Signal and TGpFileEvent.Reset.*

Signalling an event is simple (Listing 11). `TGpFileEvent.Signal` first creates the synchronisation file (which must not exist beforehand, hence the `CREATE_NEW` flag), with read access (`GENERIC_READ`). If the file already exists, `Signal` will return `true` if the file was created, `false` if the file already exists (the event is already signalled) and will raise the `EGpFileSync` exception for other problems. In any case, it will make sure that the handle to the synchronisation file is closed at the end.

This is different to what we've seen before and follows from the fact that an event will typically be reset in another process (which must be able to delete the synchronisation file, which therefore must have no open handles).

`Reset` is simple, too. If the file doesn't exist, `false` is returned; if the file exists, `Reset` will try to delete it and return `true` on success or raise an exception on failure. This looks simple, but you may have noticed a possible problem: what if `Reset` tries to delete a synchronisation file *after* it was signalled but *before* the signalling process managed to close its handle?

The short answer is that this cannot happen if you are using `TGpFileEvent` correctly. If `Reset` is called from the same thread as `Signal` then there is no problem: it is obvious that `Signal` has to close the handle and return before the same thread is able to call `Reset`. If another thread or process is waiting on this event, it should always use `WaitFor` and call `Reset` only after `WaitFor` indicates that the event is in a signalled state. And `WaitFor` (as you'll see in a moment) will make sure that the synchronisation file is not open before returning a 'signalled' result.

Of course, if you call `Reset` from a different thread or process and forgot to call `WaitFor` first, you may receive an `EGpFileSync` exception, which will kindly remind you of the error of your ways.

We still have to explain `WaitFor`, which is the toughest of the three methods and is implemented with a loop, not completely unlike the one in `TGpFileMutex.Acquire` or `TGpFileGroup.Join`. `WaitFor` takes two parameters, `timeout` (in milliseconds, `0` and `INFINITE` are supported) and `reset` (the event should be automatically reset).

First, if the event is to be automatically reset (ie the synchronisation file has to be deleted), set `FILE_FLAG_DELETE_ON_CLOSE`. The operating system will make sure the file is deleted after we're done with it.

Next, try to open the file in read-only do-not-create-if-exists mode. If the open failed because of a sharing error, go to the next step. If the open failed because of some other error, raise an exception. If the file opened without a problem, the event is signalled: call `CloseHandle` to close the open file handle (this will delete the synchronisation file if required) and return `true`.

Now check if the allotted time has elapsed. If so, return `false`.

Finally, sleep for `RetryDelay` milliseconds, then return to the previous step.

Demonstrating the use of `TGpFileEvent` are the Ping and Pong applications on the companion disk. Try starting them in a different order to see how they behave. You can even start more than one Pong and then check if each Ping will always trigger exactly one Pong (it will, I promise).

You've Got Mail

Now prepare for something completely different. All that I was talking about up till now has only been a preparation, an exercise in distributed thinking. Now I'll show you something more complicated: a file-based `SendMessage`.

It doesn't sound tough? Trust me, it is. The reason is simple: we don't want the receiver to receive a message if the sender died during the send process. Without that requirement, file messaging would be simple: the sender would write a message into a file, the receiver would read the data from the file and then delete it. Simple, yes, but that is `PostMessage`: one process

posts data and forgets about it. In `SendMessage`, we want to know that the receiver got the message. Even more, we want the receiver to know that the sender is still alive and waiting for the receiver to pick up the message. Impossible? Not with a little trickery.

Obviously, `TGpFileMessage` (see Listing 12) requires write access to the synchronisation folder, as it will transfer data from one process to another using files. It will also need three additional files for the helper synchronisation objects: one for `TGpFileMutex` and two for `TGpFileGroup`. They will use the suffixes `_lck`, `_grp`, and `_grp_lck`

Another fact worth stating is that `TGpFileMessage` does not support multicasting (sending one message to multiple recipients) because that would complicate it beyond salvation. There must be just one sender and one receiver.

To understand the `Send/Receive` mechanism, you should keep in mind that `Send` waits for `Receive` to pick the message up and only then (or if it times out, of course) does it return. `Send` and `Receive` are meant to be used in tightly coupled processes. If you need a more mailbox-like solution, you can easily build one with a file and a mutex. Mailboxes can be used in many different ways so it is probably best to make one that is tailored for your application.

The sending process is split into two parts. The `Send` method first creates a message and then waits for the message to be picked up. Receiving is also a two-part operation: the first waits for the message to appear, the second part reads it.

I already mentioned that the `Send/Receive` mechanism uses additional mutex and group primitives. The mutex is used to make some parts of `Send` and `Receive` atomic and the group is used to indicate that the sender is still alive: if `Receive` finds a waiting message and an empty group, it will know that the sender has died and will ignore the message.

```
TGpFileMessage = class(TGpFileSynchroObject)
private
  fmLock : TGpFileMutex;
  fmGroup: TGpFileGroup;
public
  constructor Create(syncFile: string); reintroduce;
  destructor  Destroy; override;
  function  Receive(timeout: DWORD; var msg: pointer; var msgSize: integer):
    boolean;
  function  Send(timeout: DWORD; msg: pointer; msgSize: integer): boolean;
end; { TGpFileMessage }
```

➤ *Listing 12: File-based messaging.*

This is the algorithm for `Send`, with some simplifications (the full implementation is shown in Figure 4):

Sending

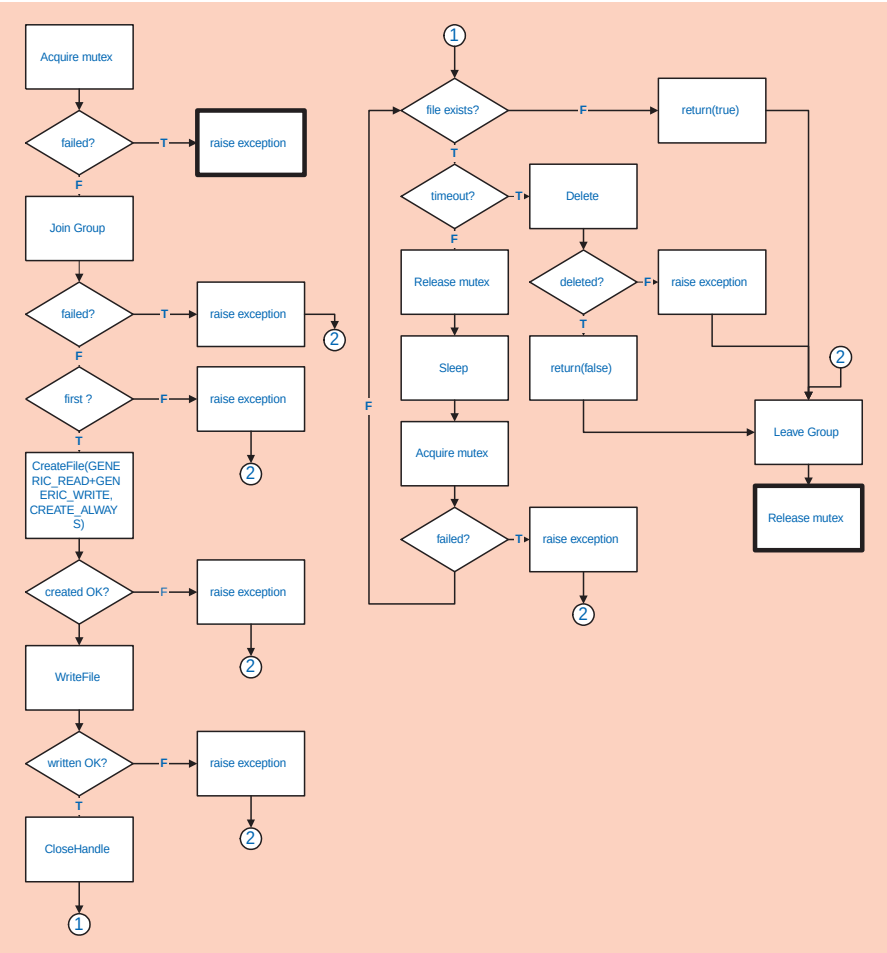
1. Acquire the mutex: message creation should be atomic. If the operation fails, abort.
2. Join the group to indicate that active sender exists. The group should be empty before joining, if it is not, there is another sender: abort.
3. Try to create the message file with read-write access and the `CREATE_ALWAYS` flag (overwrite the file if it exists). If that fails, something is terribly wrong (eg access to the synchronisation file is not allowed): abort.
4. Write message to the file and close it.

Waiting

5. Check if the message exists. If not, the receiver picked up the message, go to step 10.
6. Check if the allotted time has elapsed. If so, go to step 10.
7. Release the mutex. We must give `Receiver` some chance to read the message.
8. Sleep for `RetryDelay` milliseconds.
9. Acquire the mutex (abort if `Acquire` fails) and go to step 5.
10. Leave the group.
11. Release the mutex.

As you can see, `Send` is a member of the group all the time (to indicate that there is an active sender) and keeps the mutex acquired most of

➤ *Figure 4: Sending a message.*



the time. The mutex is released periodically in the second part of the algorithm to allow `Receiver` to read the message. Because this also allows another `Sender` to start sending using the same synchronisation file (indicating a programmer error: we assumed there is only one sender and one receiver, remember), the sender must check that the group is empty before joining it, that it is the only sender.

`Receive` works in a similar two-part manner (the full implementation is shown in Figure 5):

Waiting

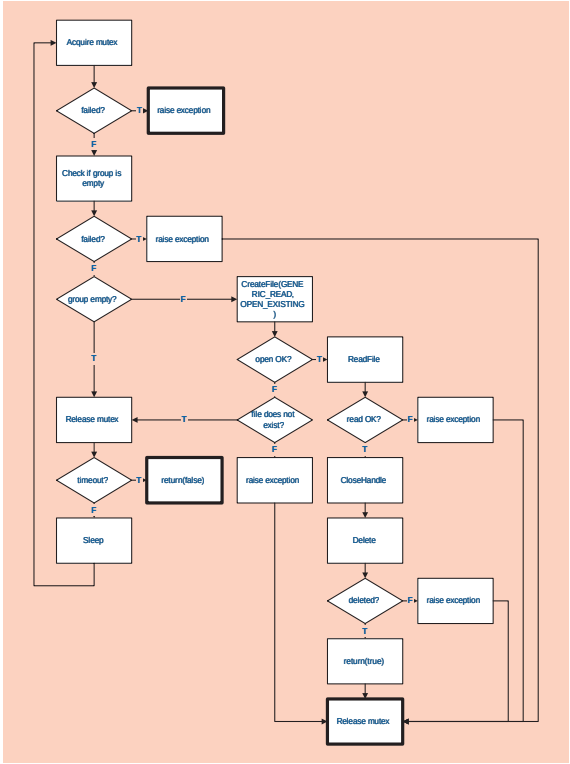
1. Acquire the mutex: message waiting must be atomic.
2. Check if the group is empty.
3. If the group is empty, either the sender has died or the receiver was started before the sender. If the group is not empty, a sender is waiting for us to pick up the message, so go to step 7.
4. Release the mutex to allow the sender to start sending the message.
5. Check if the allotted time has elapsed. If it has, return `false`.
6. Sleep for `RetryDelay` milliseconds, then return to step 1.

Reading

7. Open the message file with read access and only if it exists go on to read a message. If the file does not exist, go to step 4 (a false alarm: there is no message, return to the waiting phase).
8. Close the message file and delete it.
9. Leave the group.
10. Release the mutex.

The second part of step 7 (if the file does not exist...) requires further explanation. At first glance it is a complete nonsense: if the group is not empty and the mutex is released then surely the sender certainly has already created the message and the receiver can read it? Yes, that is true, if we send *only one* message. It is not true if the sender keeps on sending messages.

Imagine two processes. Process A will send two messages and process B will receive them. Because processes A and B run at the same time, it is possible that the actions will execute in the following order:



➤ *Figure 5: Receiving a message.*

- ➢ Process A prepares the first message and goes to sleep (send: step 8).
- ➢ At that moment, the group has one member (A) and the mutex is not acquired.
- ➢ Process B now reads the first message and deletes the message file.
- ➢ Process A is still asleep.
- ➢ Process B now starts reading the second message.
- ➢ The group is not empty and the mutex is released, so it can proceed immediately to step 8. All seems well, but the message file does not exist because the message was not sent yet.
- ➢ Process B now goes to sleep, process A awakens, sends the message and process B receives it. All ends well.

Again, I must admit that `Send` and `Receive` are very complicated, especially because of the two-part implementation and the constant acquiring and releasing of the mutex. If you want to really understand them, study the algorithms above, Figures 4 and 5, and the source code.

If you only want to use `TGpFileMessage`, take a look at the Send and Receive applications on the companion disk.

The Ultimate Challenge

Now that you've seen how complicated it can be just to send a message, I can explain why a semaphore is hard to write: so hard that I have not yet implemented it and I doubt very much that I will.

Let's summarise what a semaphore does. What follows is quite a simplification and not a very good definition of a semaphore; for better definition, see the references I mentioned at the beginning. A semaphore basically protects a counted resource. A semaphore knows how many clients are using it and allows new clients to attach only if less than some maximum value of clients (specified when the semaphore is created) are using it. When the semaphore is full, new clients must wait for old clients to stop using it before they can gain access.

This looks relatively simple and easy to implement, but there is a problem: unstable computers. Processes can die. Hardware may malfunction. Because of that, good file-system synchronisation primitives must handle sudden disappearances. All of the primitives discussed so far are very well behaved: they handle process and computer crashes quite well. But it is not so easy to write a well-behaved semaphore. I have tried several approaches and they all backfired (usually when I thought I had got it working at last!).

My current favourite idea is to use multiple files for each semaphore, as many files as the semaphore's maximum count. Each process can then allocate a semaphore by locking those files. If a process crashes, all the locked files will be released and other processes will be able to allocate them. But, as I don't often use

```
TGpFileSWMR = class
private
  fswmrGroup       : TGpFileGroup;
  fswmrMutex1      : TGpFileMutex;
  fswmrMutex2      : TGpFileMutex;
  fswmrSyncFileBase: string;
public
  constructor Create(syncFileBase: string; deleteOnRelease: boolean = false);
    reintroduce;
  destructor  Destroy; override;
  function  DoneReading(timeout: DWORD): boolean;
  procedure DoneWriting;
  function  IsReading: boolean;
  function  IsWriting: boolean;
  function  WaitToRead(timeout: DWORD): boolean;
  function  WaitToWrite(timeout: DWORD): boolean;
  property  SyncFile: string read fswmrSyncFileBase;
end; { TGpFileSWMR }
```

➤ *Listing 13: File-based Single-Writer-Multiple-Reader guard.*

semaphores, there is little chance of a `TGpFileSemaphore` appearing in public from my keyboard.

One Writer, Many Readers

To show how powerful file-based synchronisation primitives are, I will build a single-writer-multiple-reader guard (SWMR for short, see Listing 13) synchronisation class with two mutexes and one group.

SWMR is a synchronisation primitive that allows readers and writers to work with a shared resource, like a database. Multiple readers can access the shared resource at the same time, but there can only be one writer. If there is an active writer, no reading is allowed, and noone gets write access if there's an active reader.

SWMR methods are divided into two subsets: one used by readers and one by writers. `WaitToWrite` tries to acquire write access and `DoneWriting` releases it. Similarly, `WaitToRead` tries to acquire read access and `DoneReading` releases it. Because of implementation issues, `DoneReading` also requires a timeout parameter. `IsReading` and `IsWriting` are simple helper functions that return the current status.

To implement exclusive write access, the writer has to acquire a mutex (to prevent more than one process at a time getting write access). Before that, the writer has to check no readers are active. To prevent synchronisation problems, checking and acquiring must be atomic so we'll wrap them in a critical section, maintained by the second mutex.

To implement multiple read access, each reader will join a group. Before joining, the reader must check if the writer's mutex is acquired (indicating that the writer is active and reading is not permitted). To prevent synchronisation problems, checking and joining must be atomic, so we'll wrap them in a critical section, maintained by the same mutex as in the writer's case.

The `WaitToWrite` algorithm is as follows. First, acquire the 'check' mutex. See if the group is empty: if so, acquire the 'write' mutex. Next release the 'check' mutex. If the 'write' mutex is acquired, return `true`. Now check if the allotted time has elapsed: if so, return `false`. Next sleep for `RetryDelay` milliseconds, then return to the beginning.

The `WaitToRead` algorithm is as follows. First, acquire the 'check' mutex. See if the 'write' mutex is acquired: if not, join the group. Now release the 'check' mutex. If we are a member of the group, return `true`. Next check if the allotted time has elapsed: if so, return `false`. Now sleep for `RetryDelay` milliseconds, then return to the beginning.

`DoneWriting` and `DoneReading` are equally trivial: the former releases the 'write' mutex and the latter leaves the group. That's all.

Monitoring

To sugar all this I have created two monitoring components that can ease the use of most common synchronisation primitives: a mutex and a group. Both components use a background thread to monitor the synchronisation primitive and then translate changes into events. `TGpFileMutexMonitor` fires an `OnAcquired` event when some other process acquires the specified mutex and `OnReleased` when the mutex is released. `TGpFileGroupMonitor` behaves in a similar manner, except that it provides the events `OnEmpty` and `OnNotEmpty`. Both components also offer quick access to the mutex and group methods (`Acquire`, `Release`, `Join` and `Leave`), so you can also use them as a normal mutex or group.

If you want to know how these components work, check the source on the disk, where you'll also find a demonstration program testGpFileSyncMon.dpr.

The `GpFileSync` primitives are useful tools as they are programmed at the moment. But I'm expecting Kylix to open a whole new world of inter-process communication, where `GpFileSync` and its Kylix equivalent will be really, really important. When I manage to port it, of course. But more on that later: I'll be back!

Primoz Gabrijelcic has already Joined the TGpFileGroup of Kylix-waiting processes. Expect TGpFileEvent or even TGpFileMessage when he Acquires it. In the meantime, he will try to respond to all mail sent to gabr@17slon.com. You may reuse the code that accompanies this article even if you are completely unsynchronised.