A closer look at process control

Michaël Van Canneyt

March 16, 2014

1 Introduction

In the previous article about process control, the TProcess component was introduced, and an overview of its methods and properties was given, as well as a small test application which could be used to test most of the properties.

In this article, the TProcess component will be discussed in more detail: The main methods will be analyzed and explained. Then, the process of porting the component to Linux (for use with Kylix) will be explained, and some well as some small demo programs will be presented.

2 A word about pipes

Before examining the TProcess control, it is necessary to have a look at pipes. A pipe is a pair of file descriptors, which are not associated with a file on disk (not necessarily, anyway). Things that are written to one file descriptor, can be read from the other file descriptor, in a FIFO manner. This can be used to communicate between processes: First, a parent process creates a pipe (say, PP and PC), and then creates a new child process, and passes one of the file descriptors to the newly created child (PC). When the child writes to its file descriptor (PC), the parent can read what was written from the other file descriptor (PP), or vice versa.

Under Windows, the only way to pass this file descriptor to a newly created process, is to make this file descriptor one of the standard input, output or error output handlers. This must be done in the CreateProcess call.

Instead of using the pipes to communicate between processes, it would also be possible to use the pipes to communicate between threads.

To create a pipe, Windows defines the following call

```
function CreatePipe(var hReadPipe, hWritePipe: THandle;
  lpPipeAttributes: PSecurityAttributes; nSize: DWORD): BOOL; stdcall;
```

This function returns a pair of file descriptors in the hReadPipe,hWritePipe variables. The lpPipeAttributes is a pointer to a TSecurityAttributes record. The nSize parameter can be used to specifiy the size of the internal memory buffer that the pipes will use to transfer data.

Since the idea will be to pass the pipe file descriptor on to a child process, the secirity attributes of the pipe should be set so that the pipes file descriptors are passed on to a child process. For this, 2 constant TSecurityAttributes records are defined in the pipes unit:

```
Const
  piInheritablePipe : TSecurityAttributes = (
    nlength:SizeOF(TSecurityAttributes);
    lpSecurityDescriptor:Nil;
    Binherithandle:True);
  piNonInheritablePipe : TSecurityAttributes = (
    nlength:SizeOF(TSecurityAttributes);
    lpSecurityDescriptor:Nil;
    Binherithandle:False);
```

These can be used when creating pipes.

To make handling of pipes easier and more Delphi-oriented, the pipes unit implements 2 `TStream` descendents:

```
TPipeStream = Class (THandleStream)
public
  Function Seek(Offset : Longint;Origin : Word) : longint;override;
  Destructor Destroy; override;
end;

TInputPipeStream = Class(TPipeStream)
public
  Function Write(Const Buffer; Count : Longint) :Longint; Override;
end;

TOutputPipeStream = Class(TPipeStream)
Public
  Function Read(Var Buffer; Count : Longint) : longint; Override;
end;
```

These stream objects raise errors when an invalid operation is attempted:

- A seek operation (positioning of the file pointer) is not allowed on pipes. This has as a consequence that the SIZE of the pipe stream cannot be used, as the size is calculated by setting the position of stream at the and, and returning then the offset.
- A inputpipestream does not allow writing - a pipe is always one-way.
- Similar, a write pipe does not allow reading.

To create a pair of pipe streams, the following function is defined in the pipes unit:

```
Procedure CreatePipeStreams (Var InPipe : TInputPipeStream;
                             Var OutPipe : TOutputPipeStream;
                             SecAttr : PSecurityAttributes;
                             BufSize : Longint);
```

It is a simple wrapper around the Windows `CreatePipe` call, which instantiates two pipe streams with the correct handles.

The pipe stream objects will be used to redirect the input and/or output of newly created processes in the `TProcess` component.

3 A closer look at the methods of TProcess

The TProcess component is not a complicated component. It simply introduces some pascal-like properties, which will be translated to parameters for the Windows CreateProcess when the Execute method is called.

Apart from the Execute method (the most important method of TProcess), there are some other methods such as Suspend, Resume and Terminate which are simple methods to control the newly created process. These methods also are simple wrappers around Windows API calls.

The methods that set or get various properties will not be discussed, as they are self-explaining, and serve mostly to do some sanity checks, to avoid setting conflicting properties.

The first, and most important, method which will be discussed is the Execute method. The Execute method is a wrapper around the Windows CreateProcess function, which is defined as follows:

```
function CreateProcess(lpApplicationName: PChar; lpCommandLine: PChar;
  lpProcessAttributes, lpThreadAttributes: PSecurityAttributes;
  bInheritHandles: BOOL; dwCreationFlags: DWORD; lpEnvironment: Pointer;
  lpCurrentDirectory: PChar; const lpStartupInfo: TStartupInfo;
  var lpProcessInformation: TProcessInformation): BOOL; stdcall;
```

The function takes a lot of arguments, each of which is explained in detail in the Windows API reference. All arguments have been converted to one or more of the properties of the TProcess component; 2 methods exist that convert the properties of the TProcess component to field in the various parameters of the CreateProcess call; In particular the lpStartupInfo and dwCreationFlags parameters are filled with the results of the GetStartupFlags and GetCreationFlags calls. They will not be discussed here, as their functionality is a trivial setting of flags or fields.

The Execute method is implemented as follows:

```
Procedure TProcess.Execute;


Var
  PName,PDir,PCommandLine : PChar;
  FEnv : PPChar;
  FCreationFlags : Cardinal;

begin
  If poUsePipes in FProcessOptions then
    CreateStreams;
  FCreationFlags:=GetCreationFlags;
  FStartupInfo.dwFlags:=GetStartupFlags;
  PName:=Nil;
  PCommandLine:=Nil;
  PDir:=Nil;
  If FApplicationName<>'' then
    PName:=Pchar(FApplicationName);
  If FCommandLine<>'' then
    PCommandLine:=Pchar(FCommandLine);
  If FCurrentDirectory<>'' then
```

```
    PDir:=Pchar(FCurrentDirectory);
  if FEnvironment.Count<>0 then
    FEnv:=StringsToPcharList(FEnvironment)
  else
    FEnv:=Nil;
  FInheritHandles:=True;
  If Not CreateProcess (PName,PCommandLine,FProcessAttributes,FThreadAttributes,
                FInheritHandles,FCreationFlags,FEnv,PDir,FStartupInfo,
                fProcessInformation) then
    Raise Exception.CreateFmt('Failed to execute %s : %d',[FCommandLine,GetLastError
  if POUsePipes in FProcessOptions then
    begin
    FileClose(FStartupInfo.hStdInput);
    FileClose(FStartupInfo.hStdOutput);
    FileClose(FStartupInfo.hStdError);
    end;
  Fhandle:=fprocessinformation.hProcess;
  FRunning:=True;
  If FEnv<>Nil then
    FreePCharList(FEnv);
  if not (csDesigning in ComponentState) and // This would hang the IDE !
     (poWaitOnExit in FProcessOptions) and
      not (poRunSuspended in FProcessOptions) then
    WaitOnExit;
end;
```

The first thing the `Execute` method does, is checking whether pipes are needed. If so, the pipes are created in the `CreateStreams` method.

Then, the values of the properties are collected and inserted into 2 variables using the `GetCreationFlags` and `GetStartupFlags` methods. The two variables will be used in the `CreateProcess` call. After that, some manipulations are done to convert the `ApplicationName`, `CommandLine` and `Directory` properties to zero-erminated strings. If an environment was specified, then this is converted to an array of zero-terminated strings.

After that, the call to `CreateProcess` is made using the arguments collected from the various properties. If the call returned correctly, then some cleaning up is done:

- if pipes were requested, then the child end of the pipes is closed. The child has a copy of the pipe file descriptors, and closing them in the parent process doesn't affect the child's copy of the file descriptors. Failing to do so would keep the pipe open forever, even of the child has closed them.
- The array of environment strings is freed, if one was allocated.

Finally, f the `poWaitOnExit` option was specified, then a call to `WaitOnExit` is made. The call will *not* be issued in 2 cases:

1. When the component is being designed; issuing this call would freeze the IDE as long as the child process is running.
2. If `poRunSuspended` is specified: This is done as a security precaution, since the call would potentially never return, as the child process is suspended, and cannot exit, unless some external process would kill the process.

If poUsePipes was specified in the options, then pipes are created in the CreateStreams method. The pipes are created with the CreatePipeStreams method of the pipes unit:

```
Procedure TProcess.CreateStreams;

begin
  FreeStreams;
  CreatePipeStreams (FChildInputSTream,FParentOutPutStream,@piInheritablePipe,1024);
  CreatePipeStreams (FParentInputStream,FChildOutPutStream,@piInheritablePipe,1024);
  if Not (poStdErrToOutPut in FProcessOptions) then
    CreatePipeStreams (FParentErrorStream,FChildErrorStream,@piInheritablePipe,1024)
  else
    begin
    FChildErrorStream:=FChildOutPutStream;
    FParentErrorStream:=FParentInputStream;
    end;
  FStartupInfo.dwFlags:=FStartupInfo.dwFlags or Startf_UseStdHandles;
  FStartupInfo.hStdInput:=FChildInputStream.Handle;
  FStartupInfo.hStdOutput:=FChildOutPutStream.Handle;
  FStartupInfo.hStdError:=FChildErrorStream.Handle;
end;
```

Depending on the options, 2 or three pipes are created. Two for the standard input and output of the new process, and if standard error isn't redirected to standard output, a third stream is created for standard error. Note that the pipes are created with the piInheritablePipe constant of the pipes unit, so the pipe's handles will be valid in the newly created process as well.

Then the fields of the FStartupInfo record are filled in with the handles of the child end of the streams and the Startf_UseStdHandles flag is added to the dwFlags field. This will cause Windows to use the handles in the hStdInput,hStdOutput and hStdError as the handles for the standard input, output and error file descriptors of the new process.

The Suspend and Resume methods work as a pair; The first suspends execution of the new process, then second lets the process continue its execution. This mechanism works with a suspend count; Initially the suspend count of a process is zero. With each call to Suspend, the suspend count is raised with one. Each call to Resume decreases the suspend count with one; When the count reaches zero, the process continues execution. Each of the methods returns the new value of the suspend count. Their implementation is also a simple wrapper to the equivalent Windows calls:

```
Function TProcess.Suspend : Longint;

begin
  Result:=SuspendThread(ThreadHandle);
end;

Function TProcess.Resume : LongInt;

begin
  Result:=ResumeThread(ThreadHandle);
end;
```

The Windows SuspendThread and ResumeThread calls work with a Thread handle;

The main thread handle of the process (available through the `ThreadHandle` property) is used to stop the process.

To wait for the end of the process, the `WaitOnExit` call is implemented:

```
Function TProcess.WaitOnExit : Dword;

begin
  Result:=WaitForSingleObject (FprocessInformation.hProcess,Infinite);
  If Result<>Wait_Failed then
    GetExitStatus;
  FRunning:=False;
end;
```

It uses the `WaitForSingleObject` call, which can be used to wait on many things, and one of them is the end of a newly executed process.

Lastly, the `Terminate` method can be used to terminate the running process. If this call is succesfull, the progam will have stopped running, and will be removed from memory. It's implementation is a simple wrapper around the windows `TerminateProcess` call. The function returns `True` if the call was succesfull, `False` otherwise.

```
Function TProcess.Terminate(AExitCode : Integer) : Boolean;

begin
  Result:=False;
  If ExitStatus=Still_active then
    Result:=TerminateProcess(Handle,AexitCode);
end;
```

With this, the main methods of the `TProcess` component have been covered. Some more will be covered when the `Linux` version of the process is covered.

4 Porting `TProcess` to Linux

Porting the `TProcess` component to Linux, so it works with Kylix, is a three-stage process:

```
\item Porting the \file{pipes} unit.
\item Porting the \file{process} unit - the most difficult issue.
\item Porting the test program.
```

Each of these steps will be discussed in the subsequent.

Porting the pipes unit is quite easy; Pipes existed on Unix before the Win32 api was introduced, and the Microsoft implementation of pipes is quite similar to the Unix one. The definition of the unix `Pipe` call resides in the Libc unit. Thus, in the `Uses` clause of the pipes unit, the `Windows` unit must be replaced by the `Libc` unit.

To keep the interface of the unit the same across platforms, the approach was taken to keep the existing Windows interface, and just to ignore certain elements that were present in the Windows implementation, but which don't exist on windows. Since the `TSecurityAttributes` record doesn't exist on Linux, it's introduced in the `Interface` section of the unit:

```
Type
```

```
  PSecurityAttributes = ^TSecurityAttributes;
  TSecurityAttributes = Record
    nlength : Integer;
    lpSecurityDescriptor : Pointer;
    BinheritHandle : Boolean;
  end;
```

The only call that needs to be revised is the CreatePipe call. A version of this call is implemented for Linux:

```
Function  CreatePipe (Var Inhandle,OutHandle : Integer;
                      SecAttr : PSecurityAttributes;
                      BufSize : Longint) : Boolean;

Var
  Pipes : Array[1..2] of Integer;

begin
  Result:=Libc.Pipe(@Pipes[1])=0;
  If Result Then
    begin
    InHandle:=Pipes[1];
    OutHandle:=Pipes[2];
    end;
end;
```

As can be seen, the function is very easy. It just uses the Pipe call provided by the Libc unit to create the file handles. The bufsize argument is ignored (the size of a pipe buffer is a kernel parameter which cannot be changed) and the SecAttr argument is ignored as well. The BinheritHandle flag could be implemented on linux using the fcntl call to set the CloseOnExec flag of the file descriptors, but that has been omitted from the implementation, since for pipes this behaviour is not needed. The rest of the pipes unit remains unchanged.

Porting the process unit requires a little more work, but the same approach was taken as for the pipes unit: the interface of the Windows version is kept, and emulated on Linux. To be able to do this, some constants and types from the Windows unit have been declared in the interface of the process unit. The main types are:

```
TPoint
TRect
DWord
THandle
TProcessInformation
TStartupInfo
```

Some constants are also introduced, but these are not presented here as they are not needed for the understanding of the discussion. The interested reader can consult the unit source for more details.

Porting the TProcess concentrates mostly around implementing the key methods Execute, Suspend, Resume and Terminate and WaitOnExit.

The most difficult method to port is undoubtedly the Execute call. The linux implementation of this call is presented here again, with the modifications made for Linux:

```
Procedure TProcess.Execute;
```

```
Var
  FEnv : PPChar;
  FCreationFlags : Cardinal;

begin
  If poUsePipes in FProcessOptions then
    CreateStreams;
  FCreationFlags:=GetCreationFlags;
  FStartupInfo.dwFlags:=GetStartupFlags;
  if FEnvironment.Count<>0 then
    FEnv:=StringsToPcharList(FEnvironment)
  else
    FEnv:=Nil;
  FInheritHandles:=True;
  if Not CreateProcess (FApplicationName,FCommandLine,FCurrentDirectory,FEnv,
                        FStartupOptions,FProcessOptions,FStartupInfo,
                        fProcessInformation) then
    Raise Exception.CreateFmt('Failed to execute %s : %d',[FCommandLine,GetLastError
  if POUsePipes in FProcessOptions then
    begin
    FileClose(FStartupInfo.hStdInput);
    FileClose(FStartupInfo.hStdOutput);
    FileClose(FStartupInfo.hStdError);
    end;
  Fhandle:=fprocessinformation.hProcess;
  FRunning:=True;
  If FEnv<>Nil then
    FreePCharList(FEnv);
  if not (csDesigning in ComponentState) and // This would hang the IDE !
     (poWaitOnExit in FProcessOptions) and
      not (poRunSuspended in FProcessOptions) then
    WaitOnExit;
end;
```

The call is almost identical to the Windows version. The real work is done in the `CreateProcess` call. The interface of this call is almost the same as the Windows version, simply some arguments have been left out. It tries to emulate the Windows version of this call as much as possible, and used the Linux `Fork` and `Execve` calls to do this.

The `Fork` function call creates a new process, which is an exact copy of the currently running process; Parent and Child continue executing with this difference that in the child, the `Fork` call returns 0, and in the parent process, the call returns the process ID of the newly created child.

The `Execve` call replaces the currently running program with a new program; Any file descriptors that the current process has, are kept intact.

Together these two calls can be used to create a new process: First a fork is executed, and the child process replaces itself with the program to be executed. The parent program can decide to wait for the child process or not.

All this is used to implement the `CreateProcess` call as follows:

```
Function CreateProcess (PName,PCommandLine,PDir : String;
                        FEnv : PPChar;
```

```
                        StartupOptions : TStartupOptions;
                        ProcessOptions : TProcessOptions;
                        const FStartupInfo : TStartupInfo;
                        Var ProcessInfo : TProcessInformation)  : boolean;

Var
  PID : Longint;
  Argv : PPChar;
  fd : Integer;

begin
  Result:=True;
  Argv:=MakeCommand(Pname,PCommandLine,StartupOptions,ProcessOptions,FStartupInfo);
  if (pos('/',PName)=0) then
    PName:=FileSearch(Pname,GetEnv('PATH'));
  Pid:=fork;
  if Pid=0 then
   begin
   { We're in the child }
   if (PDir<>'') then
     ChDir(PDir);
   if PoUsePipes in ProcessOptions then
     begin
     dup2(FStartupInfo.hStdInput,0);
     dup2(FStartupInfo.hStdOutput,1);
     dup2(FStartupInfo.hStdError,2);
     end
   else if poNoConsole in ProcessOptions then
     begin
     fd:=FileOpen('/dev/null',fmOpenReadWrite);
     dup2(fd,0);
     dup2(fd,1);
     dup2(fd,2);
     end;
   if (poRunSuspended in ProcessOptions) then
     __raise(SIGSTOP);
   if FEnv<>Nil then
     libc.Execve(PChar(PName),Argv,Fenv)
   else
     Execv(Pchar(PName),argv);
   Halt(127);
   end
 else
   begin
   FreePcharList(Argv);
   // Copy process information.
   ProcessInfo.hProcess:=PID;
   ProcessInfo.hThread:=PID;
   ProcessInfo.dwProcessId:=PID;
   ProcessInfo.dwThreadId:=PID;
   end;
end;
```

The MakeCommand call (explained below) constructs an array of zero-terminated string from various options; This array can then be passed to execve. After that the program name is searched in the path if it does not contain an absolute pathname. Then the fork call is executed. In the child, the process changes to the startup-directory, after which the handling of standard file descriptors is taken care of:

- If poUsePipes is in the process options, then the file handles in the FStartupInfo record are copied by means of the Dup2 call to the standard input/output/error file descriptors. The Dup2 call copies one file descriptor to another; It closes the destination descriptor when necessary. After a call to Dup2, both file descriptors, referred to in the Dup2 call, point to the same physical file.
- If poNoConsole is specified in the process options, then the standard input/output/error are redirected to /dev/null; As a result, there will be no access to the console of the parent process. The effect of this call is an emulation of the poNoConsole option on Windows.

If the poRunSuspended options is specified, the child process sends itself the SIGSTOP signal. The result of this is that the child process stops execution *before* the actual process is executing. Sending this signal from the parent process would not be safe, as Linux doesn't guarantee which of the processes (parent or child) will start executing first, possibly allowing the child process already to execute before the signal is delivered. To avoid this, the child process sends itself this signal, before the actual new process is executed.

After that, the call to execve or execv is made, depending on whether an environment was specified. The Execve call is defined as follows:

```
function execve(PathName: PChar;Const argv: PPChar;Const envp: PPChar): Integer;
```

The first argument is the name of the program to be executed. This program will NOT be searched in the path. The second is a null-terminated array of null-terminated strings which represent the command-line of the program. The first element in this array is the program name. The last pointer is a pointer to a null-terminated array of null-terminated strings which contain the environment strings for the new programm.

When the call succeeds, the function will not return; if it returns, an error has happened, in which case the child process exits with exit code 127. (This is standard unix practice)

When the Fork call returns in the parent process, the parent process simply does some cleaning: It frees the argument pointer list, and copies the process ID to the fields in the ProcessInfo record. Under Windows the fields of this record will contain different values, but Linux doesn't have the concept of process or thread handles, so the process ID value is copied to all fields.

The MakeCommand call is used to construct a command-line for the program to be executed. It's implementation tries to take into account some of the options that exist in the Windows CreateProcess call:

- If the poNewConsole is specified, a xterm is spawned which will run the actual command. If the StartupOptions contain settings for the number of rows and columns of the console, this is passed on to the xterm.
- The ApplicationName is used as a title for the window which will be opened. (both x-term and application)
- Any options which set the geometry and position of the program (these can be different from the console geometry) they are passed on to the program as well.

The construction of a command-line is done using a stringlist: First of all the `CommandLine` parameter is split into its various options using the `CommandToList` function, and the various arguments are inserted in a stringlist. This function takes into consideration whitespace and quotes around arguments (or the filename).

After the construction of the initial command-line some command-line options are added or inserted in the stringlist depending on the various options, passed to the `MakeCommand` call. Finally the stringlist is converted to a null-terminated array of null-terminated strings.

All this is implemented in the `MakeCommand` function:

```
Function MakeCommand(Var AppName,CommandLine : String;
                     StartupOptions : TStartUpOptions;
                     ProcessOptions : TProcessOptions;
                     StartupInfo : TStartupInfo) : PPchar;
Const
  SNoCommandLine = 'Cannot execute empty command-line';

Var
  S  : TStringList;
  G : String;

begin
  if (AppName='') then
    begin
    If (CommandLine='') then
      Raise Exception.Create(SNoCommandline)
    end
  else
    begin
    If (CommandLine='') then
      CommandLine:=AppName;
    end;
  S:=TStringList.Create;
  try
    CommandToList(CommandLine,S);
    if poNewConsole in ProcessOptions then
      begin
      S.Insert(0,'-e');
      If (AppName<>'') then
        begin
        S.Insert(0,AppName);
        S.Insert(0,'-title');
        end;
      if suoUseCountChars in StartupOptions then
        With StartupInfo do
          begin
          S.Insert(0,Format('%dx%d',[dwXCountChars,dwYCountChars]));
          S.Insert(0,'-geometry');
          end;
      S.Insert(0,'xterm');
      end;
    if (AppName<>'') then
      begin
      S.Add(TitleOption);
```

```
      S.Add(AppName);
      end;
    With StartupInfo do
      begin
      G:='';
      if (suoUseSize in StartupOptions) then
        g:=format('%dx%d',[dwXSize,dwYsize]);
      if (suoUsePosition in StartupOptions) then
        g:=g+Format('+%d+%d',[dwX,dwY]);
      if G<>'' then
        begin
        S.Add(GeometryOption);
        S.Add(g);
        end;
      end;
    Result:=StringsToPcharList(S);
    AppName:=S[0];
  Finally
    S.free;
  end;
end;
```

The `CommandToList` and `StringsToPcharList` will not be discussed, as they are not essential to the understanding of the component.

This covers the implementation of the `Execute` call. The other calls can be translated just as straightforward:

```
Function TProcess.WaitOnExit : Dword;

begin
  Result:=WaitPid(Handle,@FExitCode,0);
  If Result=Handle then
    FExitCode:=WexitStatus(FExitCode);
  FRunning:=False;
end;
```

The `WaitPid` function waits for a process (in this case with process ID `Handle`) to terminate and returns the process ID of the terminated process. Exit information for the terminated process is returned in the `FExitCode` integer. The exit status of the process can be extracted using the `WExitStatus` function.

The `Suspend` and `Resume` functions are just as straightforward:

```
Function TProcess.Suspend : Longint;

begin
  If kill(Handle,SIGSTOP)<>0 then
    Result:=-1
  else
    Result:=1;
end;

Function TProcess.Resume : LongInt;
```

```
begin
  If kill(Handle,SIGCONT)<>0 then
    Result:=-1
  else
    Result:=0;
end;
```

The `Kill` function sends a signal to a process. The signal `SIGSTOP` causes a process to be suspended. The `SIGCONT` signal causes the process to resume it's operation. Since there is no idea of a suspend count, the dummy values 1 and zero are returned by the TProcess methods.

Finally the `Terminate` method is also implemented using a signal:

```
Function TProcess.Terminate(AExitCode : Integer) : Boolean;

begin
  Result:=False;
  Result:=kill(Handle,SIGTERM)=0;
  If Result then
    begin
    If Running then
      Result:=Kill(Handle,SIGKILL)=0;
    end;
  GetExitStatus;
end;
```

The `TERM` signal advises a program to terminate, and gives it the opportunity to clean up after itself, or even to ignore this signal. The `KILL` signal cannot be ignored, and will simply remove the process from memory. That is why both signals are sent, if needed. (a similar process happens usually for all processes when a Linux machine is shut down on the command-line)

Finally, a method which wasn't discussed earlier, but which deserves mentioning is the `GetRunning` function (the read method of the `Running` property). It's implementation on linux is slightly different from the Windows one:

```
Function TProcess.PeekLinuxExitStatus : Boolean;

begin
  Result:=WaitPID(Handle,@FExitCode,WNOHANG)=Handle;
  If Result then
    FExitCode:=wexitstatus(FExitCode)
  else
    FexitCode:=0;
end;

Function TProcess.GetRunning : Boolean;

begin
  IF FRunning then
    FRunning:=Not PeekLinuxExitStatus;
  Result:=FRunning;
end;
```

To know whether a process is still running, the `WaitPid` function can be used with the `WNOHANG` flag. This flag causes the `WaitPid` function to return at once when process is still running (the default behaviour is to wait till the process exits). This is implemented in the `PeekLinuxExitStatus` method, which returns `True` if the process exited. This result is used for the `IsRunning` call.

With this, the port of the `TProcess` call to linux is covered. The Linux code as presented here is intermixed with the Windows code through use of the conditional compilation directive `{$ifdef linux}`. What was presented above is a clean-up of this code.

Finally, the port of the test program can be considered: Apart from some invalid property errors, this didn't present any difficulties, as it was just necessary to replace the various units with their CLX counterparts, and adjusting the case of some of the unit names. Some small issues with the enabling of controls in CLX also had to be dealt with (a bug in CLX was encountered) but nothing really difficult. The interested reader can consult the code that comes on the CD-ROM.

One thing that should be noted is that the usage of pipes as it works under Windows does NOT work on Linux: This has been traced down to synchronization problems with the various threads used to copy the in and output streams of the process to the various memo's on the program's main window. When the reader and writer thread try to synchronize with the main window thread the program freezes. As a result, testing the use of the pipes doesn't work with the test program.

However, the functionality of pipes is correct. To demonstrate this, two extra examples were created. The first example is a small command-line program which sends a mail using the standard `sendmail` command: It is very simple, and is presented without further comment:

```
program testp3;

uses process,Classes;

Var
  Demo : TProcess;
  C : Char;
  S : String;

  Procedure StreamWrite(S : TStream; Line : String);

  begin
    Line:=Line+#10;
    S.Write(Line[1],Length(Line));
  end;

begin
  Demo:=TProcess.Create(Nil);
  Demo.CommandLine:='sendmail michael';
  Demo.Options:=[poUsePipes];
  Demo.Execute;
  StreamWrite(Demo.Input,'Subject: Mail to michael.');
  StreamWrite(Demo.Input,'Hello !');
  StreamWrite(Demo.input,'');
  StreamWrite(Demo.input,'This mail was sent automatically to demonstrate');
  StreamWrite(Demo.input,'The TProcess component.');
  StreamWrite(Demo.input,'');
```

```
  StreamWrite(Demo.input,'Sincerely yours,');
  StreamWrite(Demo.input,'Michael.');
  Demo.Free;
end.
```

This program demonstrates that writing to the input stream of the process works correctly. Conversely, the second demonstration program, shown in 1 shows that reading the output of a program works also correctly. The program contains a button, a Memo and a `TProcess` component. The `commandline` property is set to `'ls -l'`. The code to display the output of the `ls` call is in the `OnClick` handler of the button component:

```
procedure TForm1.Button1Click(Sender: TObject);

Var s : String;
    c : char;

begin
  MLS.Lines.Clear;
  With PLS do
    begin
    Execute;
    With Output do
      begin
      S:='';
      While (Read(C,1)=1) do
        begin
        If C<>#10 then
          S:=S+C
        else
          begin
          MLS.Lines.Add(S);
          S:='';
          end;
        end;
      If S<>'' then
        MLS.Lines.Add(S);
      end;
    WaitOnExit;
    end;
end;
```

The result of a press on the button is shown in 1.

5 Conclusion

The `TProcess` component is a small and simple component which makes running and controlling another program very easy. It's simplicity is illustrated by the fact that a port to `Linux` is not very difficult, and can be done with a remarkable degree of completeness. Although not all features can be ported, the component is ported to a large enough degree to make cross-platform development with it a realistic idea.

Figure 1: Reading the output of a program.