

# Kylix Pipes

by Jeremy Blythe

*A quick look at an underrated and possibly overlooked Inter Process Communication technique, from a new author.*

While there are many similarities between Unix and Windows that make life easier when porting Delphi code to Kylix, Kylix shouldn't be seen simply as a way of getting your Windows programs onto Linux. Linux itself has many benefits which we'll be looking at over the coming months.

One of the key Unix ideas is that many small programs can do the job of one big program. You will find that your Linux system comes with a large number of utility programs that can be used in conjunction with each other to great effect. (Take a look in /usr/bin for examples.)

As there are many programs running on your Linux box, there are many ways for them to talk to each and we'll be looking at these Inter Process Communication (IPC) techniques over the coming series of articles.

## Shell pipes

We'll start by using a standard Linux utility from a Kylix program.

As we all know, a common task is searching for text in a file or group of files; you will have used the 'Find in Files' feature of the Delphi IDE. This function can easily be added to a Kylix program simply by using the 'grep' Linux utility. If you type

```
grep -help
```

at the command line you will get concise information about the use of the utility. As an example, to find all the lines containing the word 'else' in any pas files in the current directory you could type:

```
grep -i -n 'else' ./*.pas
```

The -i tells 'grep' to ignore case and -n shows line numbers in the output:

```
./uMain.pas:130:   else
./uMain.pas:165:   else
./uScript.pas:68:   else if c = '/'
                    then
./uScript.pas:77:           else
./uScript.pas:80:           else
./uScript.pas:83:   else if not
                    InComment and (c = ';')
                    then
./uScript.pas:89:   else if not
                    InComment then
./uScript.pas:122:  else if (s[i] =
                    #13) or (s[i] = #10) or
                    (s[i] = #0) then
./uScript.pas:124:  else
```

So, all we need to do is to run 'grep' from our Kylix program and display the result. This is done using pipes. You may have used pipes before in MSDOS or from a Windows command prompt. This is the | operator. For example, if the output from a command is too long to fit on your

screen, you can pipe the output through 'more'. 'more' is a simple utility which takes text input and pauses the display after each screenful. So with the command: 'dir | more', the output of 'dir' is fed through the pipe into 'more', and 'more' then displays the text.

Pipes can be used inside a program as well. We need to run 'grep' from our program and pipe the output back into it, so that we can display the results. Linux provides the functions 'popen' and 'pclose' to do this job and Kylix has them in the LibC unit:

```
function popen(const Command: PChar;
               Modes: PChar): PIOFile;
function pclose(Stream : PIOFile):
               integer;
```

Below is the code (from a button OnClick event) which writes the results from 'grep' into a memo. The user will have entered the directory path and search text into edit boxes for us, which we then pass out to 'grep'.

```
procedure TForm1.Button1Click(Sender:
                           TObject);
var
  pFile : PIOFile;
  Buffer : array [0..BUFSIZ] of char;
  nChars, i : integer;
  Cmd : array [0..1023] of char;
  s : string;
begin
  Memo1.Lines.Clear;
  strcpy(Cmd, 'grep -i -n
          '+Edit1.Text+'
          '+Edit2.Text);
  pFile := popen(Cmd,'r');

  if Assigned(pFile) then
  begin
    nChars := fread(@buffer[0],
                   sizeof(char), BUFSIZ,
                   pFile);
    while nChars > 0 do
    begin
      for i := 0 to nChars-1 do
        s := s + Buffer[i];
      Memo1.Text := Memo1.Text + s;
      nChars := fread(@buffer[0],
                     sizeof(char), BUFSIZ,
                     pFile);
    end;
    pclose(pFile);
  end;
end;
```

Our program is receiving data through the pipe. We set the 'open\_mode' to 'r' meaning the data is read only, before outputting the result to the memo.

We can also use ‘popen’ to send data to another application by setting ‘open\_mode’ to ‘w’. The example below sends text through a pipe to a Linux utility that shows the hex version of the data.

```
procedure TForm1.Button1Click(Sender:
    TObject);
var
    fp : PIOFile;
    buffer : array [0..BUFSIZ] of char;
begin
    strcpy(buffer, Edit1.Text);
    fp := popen('hexdump -v', 'w');
    if Assigned(fp) then
        begin
            fwrite(@buffer[0], sizeof(char),
                length(Edit1.Text), fp);
            pclose(fp);
        end;
    end;
end;
```

Unfortunately, shell pipes aren’t amazingly efficient. Every time you make a call to ‘popen’ the system has to open a new shell and run the command in it. You won’t notice any lack of speed in these examples because the pipe is used once to create the final result. If you wanted to use pipes for continuous communication, then doing this through the shell is not as appropriate. If the two communicating programs were always running, waiting for requests or sending them, then a different type of pipe is required. This type of communication can be achieved with ‘Named Pipes’.

## Named Pipes (or FIFOs)

A ‘Named Pipe’ takes in another Unix idea. In Unix everything is a file. If you have a look in ‘/dev’ you will find filenames for all the devices in your machine. We can set up one of these special files as a ‘Named Pipe’. That way we’ll have a known communication channel that two applications can use. We can use the command line to create the pipe.

```
mkfifo /tmp/testpipe
```

We can see this pipe in the file system with the ‘ls’ command:

```
ls -lF /tmp/testpipe
prw-r--r-- 1 root root
0 Nov 10 10:38 /tmp/
testpipe|
```

The ‘p’ at the beginning and the | at the end indicate that this is a pipe. We can use this pipe on the command line as well as in programs, for example:

```
[root@ludwig /tmp]# cat < /tmp/
testpipe &
[1] 1183
[root@ludwig /tmp]# echo "hello" > /
tmp/testpipe
[root@ludwig /tmp]# hello

[1]+  Done                  cat </
tmp/testpipe
```

The first command redirects the output of ‘/tmp/testpipe’ to ‘cat’ (a utility that simply echoes its input to the screen),

the ‘&’ operator makes this command run in the background. This allows us to put something in the pipe with the second command. As you can see, ‘hello’ was put in the pipe by using ‘echo’ and has been printed to the screen by ‘cat’.

The following two programs perform similar tasks to the command lines above. One sends data to the pipe and the other receives the data. Traditionally these programs are known as producers and consumers. The producer has an edit box where you can type text to send. The consumer simply has a memo which displays the data received through the pipe.

### Producer

```
const
    FIFO_NAME = '/tmp/testpipe';
    BUFFER_SIZE = 512;

procedure TForm1.Button1Click(Sender:
    TObject);
var
    hPipe : integer;
    Buffer : array [0..BUFFER_SIZE-1] of
        char;
begin
    if access(FIFO_NAME, F_OK) = -1 then
        MessageDlg('Could not access
            '+FIFO_NAME, mtError,
                [mbOK], 0)
    else
        begin
            hPipe := open(FIFO_NAME,
                O_WRONLY);
            if hPipe = -1 then
                MessageDlg('Could not open
                    '+FIFO_NAME, mtError,
                        [mbOK], 0)
            else
                begin
                    strcpy(Buffer, Edit1.Text);
                    if __write(hPipe, Buffer,
                        length(Edit1.Text)) = -1
                        then
                            MessageDlg('Write error',
                                mtError, [mbOK], 0)
                        else
                            __close(hPipe);
                end;
            end;
        end;
end;
```

### Consumer

```
const
    FIFO_NAME = '/tmp/testpipe';
    BUFFER_SIZE = 512;

procedure TForm1.Button1Click(Sender:
    TObject);
var
    Buffer : array [0..BUFFER_SIZE-1] of
        char;
    nRead : integer;
    s : string;
    i : integer;
begin
    hPipe := open(FIFO_NAME, O_RDONLY);
    if hPipe = -1 then
        Mem1.Text := 'Could not open
            '+FIFO_NAME
```

```

else
begin
  repeat
    nRead := __read(hPipe, Buffer,
      BUFFER_SIZE);
    if nRead > 0 then
      begin
        for i := 0 to nRead-1 do
          s := s + Buffer[i];
          Mem01.Text := Mem01.Text + s;
        end;
      until nRead <= 0;
      __close(hPipe);
    end;
  end;
end;

```

If we run these two programs and click the 'send' button on the Producer, the program appears to hang. It does this until we click the 'receive' button on the Consumer. This is because we have opened the pipe in 'full blocking' mode. The Producer opens the pipe for writing and waits for it to be opened for reading by another application. Likewise if we clicked on receive before sending anything, the Consumer would be waiting for the Producer to open the pipe for writing. As you can see from the code, there is nothing really special going on here; the applications are simply using files to read or write. (The `__read`, `__write` and `__close` functions have the double underscores to distinguish these LibC functions from the standard System ones.)

## What about Windows?

Pipes exist in Windows as well. You can certainly set up named pipe communication using the `WaitNamedPipe`, `CreateFile`, `SetNamedPipeHandleState`, `WriteFile` and `ReadFile` Windows API functions. Also if you look carefully in your Delphi bin directory you might find a command line utility program called 'grep'. I wonder what that's for?

---

*Jeremy Blythe has been programming in Delphi since version 1 and Turbo Pascal before that. His Unix skills are from University, and a lot of late nights with Linux! Jeremy is currently a senior software developer for Pharos Communications in Reading. His email address is: [jeremy@pharos-comms.tele2.co.uk](mailto:jeremy@pharos-comms.tele2.co.uk) if you wish to contact him.*

## Squid'll Fix It

Keep your codesite statements separate from the rest of your code by starting them in column 81.

That keeps them out of the way during normal editing and stops them from being printed on your hard copy (if you print without line wraps).

*Adrian Rye*