COPIA LITERAL

Extracto del libro “GAMBAS, programación visual con Software Libre”, de la editorial EDIT LIN EDITORIAL S.L., cuyos autores son Daniel Campos Fernández y José Luis Redrejo.

LICENCIA DE ESTE DOCUMENTO

Se permite la copia y distribución de la totalidad o parte de esta obra sin ánimo de lucro. Toda copia total o parcial deberá citar expresamente el nombre del autor, nombre de la editorial e incluir esta misma licencia, añadiendo, si es copia literal, la mención “copia literal”. Se autoriza la modificación y traducción de la obra sin ánimo de lucro siempre que se haga constar en la obra resultante de la modificación el nombre de la obra originaria, el autor de la obra originaria y el nombre de la editorial. La obra resultante también será libremente reproducida, distribuida, comunicada al público y transformada en términos similares a los expuestos en esta licencia.

4. Gestión de procesos

4. Gestión de procesos
4. Gestión de procesos
4. Gestión de procesos

4.1 La ayuda ofrecida por otros programas

Los sistemas GNU/Linux siguen la filosofía de UNIX. Parte de ella consiste en crear pequeños programas especializados en cada tarea, en lugar de generar grandes aplicaciones monolíticas. Como resultado, en GNU/Linux disponemos de muchas pequeñas utilidades de consola capaces de desarrollar casi cualquier tarea que necesitemos. Los programas con interfaz gráfica, habitualmente, son simples *front-ends* para aplicaciones de línea de comandos. Podemos poner como ejemplo el magnífico programa de grabación de CDs y DVDs *K3B*, una aplicación sencilla, bella e intuitiva que, sin embargo, es sólo la portada de aplicaciones como *cdrecord* o *mkisofs*, potentes herramientas de consola. Reproductores de vídeo o audio como *Totem* o *Kmplayer*, recubren también aplicaciones sin interfaz gráfica propia, como el gran *Mplayer*.

Para aquellos que provienen del entorno Win32/VB, puede resultar extraño, acostumbrados a trabajar sólo con los recursos que aporta el propio entorno de programación o añadiendo *llamadas a la API*, esto es, trabajando con librerías del sistema cuando VB se queda corto, pero este modelo de segmentación en pequeñas unidades ofrece pronto grandes ventajas al programador.

GNU/Linux permite desarrollar sin reinventar la rueda. La shell *bash*, común en estos sistemas, junto con las herramientas habituales de consola que acompañan a cualquier distribución, proporcionan todo lo necesario para grabar un CD, reproducir vídeo, gestionar servicios LDAP, transmitir ficheros con ftp, http, smb, scp o nfs, enviar y recibir correos, administrar bases de datos, convertir formatos de ficheros gráficos o de texto y muchas tareas más. No es necesario, en la mayor parte de nuestros desarrollos, entrar en las complejidades de las diversas librerías escritas en C, tratar de solucionar los problemas de conversión de tipos entre un lenguaje de alto nivel y C, ni caer frecuentemente en *violaciones de segmento* por un descuido en un puntero mal gestionado. Basta con consultar la documentación de un comando y llamarlo tal y como haríamos manualmente desde la consola del sistema.

La depuración del programa también resulta más sencilla: basta probar el comando directamente desde un terminal de texto y comprobar los resultados, antes de embeberlo en el código Gambas.

4. 2 Gestión potente de procesos

A diferencia de VB, donde lo único que podíamos hacer sin ayuda de la API era lanzar un proceso y desentendernos de él, Gambas permite sincronizar la ejecución de los dos programas, comunicarse con él leyendo y escribiendo por la entrada y salida estándar (*stdin* y *stdout*), conocer el estado (en ejecución o finalizado) y recibir sus mensajes de error por la salida estándar de errores (*stderr*).

4. 3 EXEC

Existen dos comandos para lanzar la ejecución de programas desde Gambas: *EXEC* y *SHELL*. Un programa en ejecución se denomina proceso y a partir de ahora hablaremos de procesos más que de programas. La primera instrucción, EXEC, lanza el comando que indiquemos, acompañado de los parámetros que escribamos:

```
[Variable=] EXEC [Command][ WAIT ][ FOR
(READ|WRITE|READ WRITE) [TO String]
```

Para facilitar la escritura de los parámetros, evitando los problemas que pueden surgir con caracteres especiales tales como los espacios, la sintaxis de *EXEC* indica que hemos de pasar el comando y los parámetros como una matriz o array de cadenas:

```
[Command,param1,param2...]
```


De esta forma, si tenemos que ejecutar, por ejemplo, el comando *cat* con un nombre de archivo tal como *mi archivo.txt*, que tiene un espacio en medio, desaparece la ambigüedad y la necesidad de indicar más caracteres especiales como \, con el fin de eliminar la posibilidad de que el comando tome *mi archivo.txt* como dos parámetros en lugar de uno sólo.

Vamos a ir desgranando poco a poco las diferentes opciones. *Command* es el único parámetro obligatorio y representa el comando a ejecutar. En el modo de trabajo más simple de *EXEC*, éste ejecuta el comando que le indiquemos y se desentiende de él. Ahora, vamos a crear un proyecto de consola con Gambas. Para ello, añadimos un módulo de inicio y, como único código, escribimos:

```
PUBLIC SUB Main()
  EXEC ["ls","-l"]
END
```

Al ejecutarlo se lanza el comando **ls** con el parámetro **-l**, obteniendo un listado de los archivos de la carpeta actual en formato largo. Observemos que nuestra matriz de cadenas está indicada directamente en el comando; podríamos haber hecho también el programa así:

```
PUBLIC SUB Main()

  DIM sCad AS NEW String[]

  sCad.Add("ls")
  sCad.Add("-l")
  EXEC sCad
END
```

Pero el intérprete de Gambas es capaz de reconocer una matriz indicando las cadenas entre corchetes y, de esa forma, nos hemos librado de unas cuantas líneas de código.

Hasta aquí todo lo que hace Gambas es ejecutar el nuevo proceso, pasándole los parámetros, y desentenderse de él. Se ejecuta de forma asíncrona, es decir, el programa Gambas sigue su curso sin esperar a que el proceso hijo finalice. Esto puede ser un inconveniente si tenemos que sincronizar, o esperar a que el proceso acabe, antes de continuar con la siguiente instrucción. Podemos realizar esta tarea de las tres maneras que vemos a continuación.

Palabra clave WAIT

Si añadimos el flag *WAIT* a la instrucción *EXEC*, el programa principal se detendrá hasta que el proceso haya finalizado de forma normal o debido a algún fallo.

Veamos de nuevo el primer ejemplo con pequeñas modificaciones. Vamos a hacer un listado de la carpeta **/dev**, que contiene gran cantidad de archivos. Si los dos procesos se ejecutan de forma asíncrona, obtendremos resultados impredecibles:

```
PUBLIC SUB Main()
  EXEC ["ls","/dev","-l"]
  PRINT "HOLA DESDE GAMBAS"
END
```

A continuación, lo compilamos y ejecutamos varias veces desde un terminal. Observaremos que, como en el listado de más abajo, la frase HOLA DESDE GAMBAS se introduce de forma caprichosa entre el listado generado por el comando ls:

```
...
rwxrwxrwx  1 root  root  5 jun 28 10:55 mouse -> psaux
crw-rw----  1 root  root  13,  32 jun 28 10:55 mouse0
drwxr-xr-x  2 root  root  60 jun 28 10:55 net
crw-rw-rw-  1 root  root  1,  3 jun 28 10:55 null
lrwxrwxrwx  1 root  root  3 jun 28 10:55 par0 -> lp0
HOLA DESDE GAMBAS
crw-rw----  1 daniel usb  99,  0 jun 28 10:55 parport0
crw-rw----  1 root  root  10,  62 jun 28 10:55 pktcdvd
crw-r-----  1 root  root  1,  4 jun 28 10:55 port
crw-------  1 root  root  108,  0 jun 28 10:55 ppp
...
```

Apliquemos ahora el flag WAIT:

```
PUBLIC SUB Main()
  EXEC ["ls","/dev","-l"] WAIT
  PRINT "HOLA DESDE GAMBAS"
END
```

Lo ejecutamos cuantas veces queramos: ya no existe el problema inicial, el programa Gambas espera a que termine de ejecutarse *ls*, y después pasa a la siguiente línea de código. Hemos conseguido sincronizar la ejecución de dos procesos de forma sencilla, simplificando nuestro código, ya que de otro modo tendríamos, por ejemplo,

que haber volcado el listado en un fichero, esperar en un bucle a que el tamaño del fichero dejase de crecer y, a continuación, leerlo y mostrarlo en pantalla.

El descriptor del proceso

Habremos observado que al principio de la sintaxis de *EXEC*, se indica un valor opcional *VARIABLE=*, que recibe algo de retorno tras llamar a *EXEC*. Esta variable es un objeto de la clase *Process*, y lo que se recibe es un descriptor del proceso que hemos lanzado. Los objetos de la clase *Process* tienen una serie de propiedades que nos permiten conocer el estado del proceso, así como actuar sobre él.

Lo que nos interesa ahora es la propiedad *State*, que refleja el estado de ejecución. Cuando se lanza un proceso, el valor de *State* es *Process.Running*, es decir, proceso en ejecución. Si el programa ya ha terminado, la variable *State* podrá tomar dos valores *Process.Stopped*, detenido, finalizado, o *Process.Crashed*, si finalizó debido a un error grave, habitualmente una violación de segmento.

Utilizando esta propiedad tendremos la capacidad de sincronizar los dos procesos de un modo más potente: podremos realizar algunas tareas en nuestro programa Gambas, mientras esperamos a que finalize el proceso auxiliar. Típicamente, lo que haremos será dar algo de feedback o información al usuario de que debe esperar. Como ejemplo, vamos a descargar un fichero desde Internet cuando el usuario pulsa un botón, y a indicarle que estamos trabajando, que no se ha colgado la aplicación, si no que debe esperar con paciencia.

Vamos a trabajar con la aplicación auxiliar *curl*, que es un programa de línea de comandos que permite precisamente lo que queremos: descargar un fichero desde una URL. Si no tenemos *curl* ya instalado, aprovecharemos para hacerlo ahora, ya que está disponible para todas las distribuciones GNU/Linux habituales, así como para FreeBSD. Para ello, consultamos desde Synaptic, Yast, RpmDrake o nuestro gestor habitual de paquetes en otras distribuciones.

Lo que descargaremos es un programa en Gambas llamado *RadioGambas*, cuyo código contiene un buen ejemplo de gestión de procesos y que también podemos estudiar.

Además, nos sirve para escuchar programas de radio emitidos por Internet, lo cual no está mal. La URL es **http://gambas.gnulinex.org/radiogambas/RadioGambas-1.0.1.tar.gz**, aunque podemos consultar en **http://gambas.gnulinex.org/radiogambas** la existencia de versiones más recientes.

Figura 1. Valor **Inactivo** de la etiqueta y botón **Descargar**.

Ahora, vamos a crear un programa gráfico y, en él, un formulario, con un botón llamado **BtnDescarga** y una etiqueta llamada **LblInfo**.

La etiqueta tendrá como propiedad **Text** el valor **Inactivo**, y el botón tendrá el texto **Descargar**.

El código del formulario será el siguiente:

```
PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process
  DIM sUrl AS STRING


  sUrl="http://gambas.gnulinex.org/radiogambas/
  RadioGambas-1.0.1.tar.gz"
  hProc = EXEC ["curl", sURL, "-o", User.Home &
  "/RadioGambas.tar.gz"]

  DO WHILE hProc.State = Process.Running
    SELECT CASE LblInfo.Text
      CASE "|"
        LblInfo.Text = "/"
      CASE "/"
        LblInfo.Text = "-"
      CASE "-"
        LblInfo.Text = "\\"
```

```
        CASE "\\"
          LblInfo.Text = "|"
        CASE ELSE
          LblInfo.Text = "|"
      END SELECT
      WAIT 0.1
    LOOP

    LblInfo.Text = "Inactivo"
    Message.Info("Descarga finalizada")

  END
```

Ejecutamos el proceso con los parámetros necesarios para que lo descargue en nuestra carpeta personal con el nombre **RadioGambas.tar.gz**, y recibimos un descriptor del proceso. Seguidamente, entramos en un bucle que se ejecuta mientras el proceso está vivo, es decir, mientras el estado del proceso es **Process.Running**. En dicho bucle cambiamos el valor del texto de la etiqueta entre los valores "-", "|", "/" y "\", de forma que se genera la ilusión de un molinillo, con lo que el usuario sabe que algo se está cociendo por debajo. Con la instrucción **WAIT** refrescamos la interfaz. Al terminar, informamos al usuario y ponemos la etiqueta con su valor original.

Probamos a modificar el programa eliminando el código del molinillo y ejecutando el programa de forma síncrona, como explicamos con el flag *WAIT*:

```
hProc=EXEC["curl", sUrl , "-o", System.Home &
"/RadioGambas.tar.gz"] WAIT
```

El funcionamiento es igual de efectivo: el programa descarga el fichero en ambos casos, pero ahora la interfaz de usuario queda bloqueada durante la descarga, lo que puede hacer pensar que nuestro programa se ha colgado y generar algunas llamadas

inútiles a nuestro servicio de atención al cliente, en el peor de los casos. Es importante, por ello, evaluar en qué casos nos conviene usar *WAIT* y en cuáles es mejor informar, de alguna manera, al usuario para que mantenga la calma y las manos lejos del teléfono.

Redirección con TO

En los ejemplos anteriores con el comando *ls*, la salida aparecía directamente en la consola, lo cual no es útil si queremos procesar la información procedente del comando.

Podemos utilizar la palabra clave TO para conseguir dos propósitos de forma sencilla: esperar a que el proceso acabe antes de continuar el programa principal y recibir en una cadena de texto la salida del programa:

```
PUBLIC SUB Main()

  DIM sCads AS NEW String[]
  DIM Buf AS String
  DIM Bucle AS Integer

  EXEC ["ls", "-l"] TO Buf

  sCads = Split(Buf, "\n")
  sCads.Remove(0)
  sCads.Remove(sCads.Count - 1)

  FOR Bucle = 0 TO sCads.Count - 1
    PRINT Left(sCads[Bucle], 10)
  NEXT

END
```

La instrucción EXEC aguarda hasta que finalice el comando **ls**, almacenando en un buffer la salida estándar del comando, que nos devuelve en la cadena **Buf**. A continuación, procesamos la cadena **Buf** separándola en líneas con **Split**, eliminamos la primera y última (sin información útil) y mostramos en pantalla sólo la parte del listado correspondiente a los permisos.

La salida de los procesos con varias líneas se puede dividir fácilmente empleando la función *Split*, y utilizando como separador el retorno de carro *\n*.

Hasta aquí hemos visto cómo sincronizar los dos procesos, pero aún podemos tener más control sobre él.

Matar un proceso

Además de las propiedades del objeto *Process*, éste ofrece un método de gran importancia: *Kill*, el cual permite matar o acabar con el proceso en cualquier momento. Supongamos que en nuestro programa anterior, la red es desesperadamente lenta y el usuario decide no esperar e interrumpir la descarga. Gracias a *Kill* podemos incluir esa posibilidad en nuestro programa, vamos a ver cómo llevarlo a cabo.

Añadimos al programa anterior un botón llamado **BtnCancelar**, con texto **Cancelar** y con el flag **Enabled** a FALSE, para que inicialmente esté inactivo.

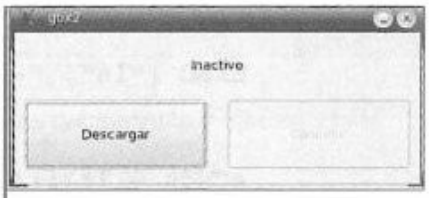

Figura 2. **BtnCancelar Inactivo**.

El nuevo código es el siguiente:

```
PRIVATE hCancel AS BOOLEAN

PUBLIC SUB BtnCancelar_Click()
  hCancel=TRUE
END
```

```
PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process
  DIM sUrl AS String

  hCancel=FALSE
  BtnCancelar.Enabled=TRUE
  sUrl="http://gambas.gnulinex.org/radiogambas/
  RadioGambas-1.0.1.tar.gz"
  hProc = EXEC ["curl", sUrl, "-o", User.Home &
  "/RadioGambas.tar.gz"]

  DO WHILE hProc.State = Process.Running
    SELECT CASE LblInfo.Text
      CASE "|"
        LblInfo.Text = "/"
      CASE "/"
        LblInfo.Text = "-"
      CASE "-"
        LblInfo.Text = "\\"
      CASE "\\"
        LblInfo.Text = "|"
      CASE ELSE
        LblInfo.Text = "|"
    END SELECT
    WAIT 0.1
    IF hCancel=TRUE THEN
      hProc.Kill()
      Message.Warning("Proceso cancelado")
      LblInfo.Text = "Inactivo"
      BtnCancelar.Enabled=FALSE
      RETURN
    END IF
```

```
    LOOP

    LblInfo.Text = "Inactivo"
    Message.Info("Descarga finalizada")

END
```

Disponemos de una variable global, **hCancel**, para saber si el usuario ha pulsado el botón **Cancelar**. Al iniciar la descarga, ponemos esta variable a **FALSE**. Si el usuario pulsa el botón **BtnCancelar**, dicha variable toma el valor **TRUE**. En nuestro bucle, comprobamos el valor de **hCancel** y, de ser **TRUE**, matamos el proceso con **Kill()**, devolvemos la interfaz al estado inactivo, informamos al usuario y salimos de la subrutina.

Aunque ya tenemos sincronización del proceso y podemos acabar con él a nuestro antojo, el feedback que hasta aquí hemos proporcionado al usuario es algo pobre: el molinillo no nos sirve para conocer cuál es el estado real de la descarga, ni hacernos idea de cuánto más habremos de esperar.

Sin embargo, *curl* está emitiendo un informe por la salida estándar de errores *stderr* (o texto impreso por la consola, si no estamos familiarizados con ese término), que podemos aprovechar.

Redirección de la salida estándar de errores

Si estamos familiarizados con el lenguaje C, los mensajes se envían a la salida estándar con la función *printf()* y a la salida estándar de errores mediante *perror()*, ambas incluidas en la librería estándar de C (*glibc* en sistemas GNU/Linux).

Los procesos pueden enviar texto a la consola por dos vías, la primera es utilizando la salida estándar y, la segunda, la salida estándar de errores. Los dos caminos separados se utilizan para diferenciar qué tipos de mensajes se envían. Por la salida

estándar (*stdout*) se suele emitir información útil. Como veremos en el siguiente apartado, el programa *curl* emite el fichero recibido por la salida estándar salvo que indiquemos expresamente el fichero dónde depositarla. Por la salida estándar de errores (*stderr*) se emiten mensajes de estado, de advertencia o de error. *Curl* aprovecha esta salida para indicar el estado de la descarga o para informar de un error en el intento de conexión.

Centrándonos más en *curl*, si aplicamos el parámetro -#, el programa va mostrando una barra de progreso formada por el símbolo # (almohadilla o sostenido) y un indicador del tanto por ciento descargado.

Gambas permite recoger el contenido tanto de la salida estándar como de la salida estándar de errores, si aplicamos el flag *FOR READ* a la hora de ejecutarlo.

Cuando el proceso hijo envía una cadena por la salida estándar de errores (llamando a *perror()*, si está escrito en C), nuestro programa Gambas recibirá un evento **Error()** procedente de la clase **Process**. La sintaxis de este evento es:

```
PUBLIC SUB Process_Error(sError As String)
  ...
END
```

En la cadena **sError** recibiremos el mensaje de error, o informativo, procedente del proceso hijo. Vamos entonces a modificar el programa para recibir el porcentaje de descarga y representarlo en la etiqueta, en lugar del molinillo.

```
PRIVATE hCancel AS Boolean

PUBLIC SUB BtnCancelar_Click()
  hCancel = TRUE
END
```

```
PUBLIC SUB Process_Error(Err AS String)

  DIM sCad AS String[]

  Err = Trim(Err)
  sCad = Split(Err, " ")
  LblInfo.Text = sCad[sCad.Count - 1]

END

PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process
  DIM sUrl AS String

  hCancel = FALSE
  BtnCancelar.Enabled = TRUE
  sUrl="http://gambas.gnulinex.org/radiogambas/
  RadioGambas-1.0.1.tar.gz"
  hProc = EXEC ["curl", , "-o", User.Home &
  "/RadioGambas.tar.gz", "-#"] FOR READ

  DO WHILE hProc.State = Process.Running
    WAIT 0.1
    IF hCancel = TRUE THEN
      hProc.Kill()
      Message.Warning("Proceso cancelado")
      LblInfo.Text = "Inactivo"
      BtnCancelar.Enabled = FALSE
      RETURN
    END IF

  LOOP
```

```
    LblInfo.Text = "Inactivo"
    Message.Info("Descarga finalizada")

END
```

En esta ocasión ejecutamos el programa **curl** con el parámetro adicional -#. Cada vez que nuestro programa recibe un evento **Error de Process**, éste se trata en nuestro código: tomamos la cadena, eliminamos con **Trim()** los posibles caracteres especiales de control que *curl* usa para mantener el cursor siempre en la misma línea, de modo que dé la impresión que la barra de progreso avanza (se trata de los llamados caracteres ANSI de control), separamos la cadena en varias subcadenas tomando el carácter de espacio como base y nos quedamos con la última subcadena, que es la que contiene el dato del porcentaje, para representarla en la etiqueta.

Redirección de la salida estándar

Supongamos ahora que no deseamos guardar un fichero, sino mostrar directamente su contenido en pantalla. Por ejemplo, en la URL **http://www.gnu.org/licenses/gpl.txt**, disponemos de un fichero que, en texto plano, contiene la licencia GPL original en inglés.

La primera opción sería guardar el fichero y, una vez finalizado el proceso, leerlo y representarlo en pantalla, pero de este modo complicamos el código ya que tenemos que crear un fichero en alguna ubicación (por ejemplo, */tmp*), leerlo y luego borrarlo para no dejar restos en el disco duro. Como indicamos en el anterior apartado, el contenido de la salida estándar también puede ser leído y *curl* envía el fichero a la salida estándar salvo que, como en nuestros ejemplos anteriores, especifiquemos un fichero donde depositar los datos recibidos. La sintaxis del evento generado por la recepción de datos procedentes de la salida estándar del proceso, es algo diferente de la de los erorres o informativos que vimos antes:

```
PUBLIC SUB Process_Read()
  ...
END
```

En este caso no hay ninguna cadena para recibir la información, por el contrario, cada objeto **Process** se comporta como un flujo o *stream*, lo que en otras palabras significa que podemos trabajar con él del mismo modo que haríamos con ficheros abiertos con *OPEN*, pudiendo utilizar, por tanto, **READ** o **LINE INPUT**.

Hay que recordar también que Gambas provee una palabra clave, *LAST*, que dentro del evento representa de forma genérica al objeto que lo generó. Podemos, entonces, leer el contenido de la salida estándar del proceso utilizando *LAST* como parámetro de las instrucciones relacionadas con lectura y escritura de procesos. Por ejemplo, para leer una línea completa, haríamos:

```
PUBLIC SUB Process_Read()

  DIM sCad AS STRING

  LINE INPUT #LAST,sCad
  PRINT sCad

END
```


Para nuestro programa, añadiremos una caja de texto (*TextArea*), llamándola **TxtLicencia**, con su texto inicial en blanco y en ella pondremos el contenido del fichero **gpl.txt**, conforme lo recibimos.

Figura 3. Caja de texto **TxtLicencia**.

El código es el siguiente:

```
PRIVATE hCancel AS Boolean

PUBLIC SUB BtnCancelar_Click()
  hCancel = TRUE
END
```

```
PUBLIC SUB Process_Read()

  DIM sCad AS String

  LINE INPUT #LAST, sCad
  TxTLicencia.Text = TxTLicencia.Text & sCad & "\n"

END

PUBLIC SUB Process_Error(Err AS String)

  DIM sCad AS String[]

  Err = Trim(Err)
  sCad = Split(Err, " ")
  LblInfo.Text = sCad[sCad.Count - 1]

END

PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process

  TxtLicencia.Text = ""
  hCancel = FALSE
  BtnCancelar.Enabled = TRUE
  hProc = EXEC ["curl", "http://www.gnu.org/licenses/
  gpl.txt", "-#"] FOR READ

  DO WHILE hProc.State = Process.Running
    WAIT 0.1
    IF hCancel = TRUE THEN
      hProc.Kill()
```

```
        Message.Warning("Proceso cancelado")
        LblInfo.Text = "Inactivo"
        BtnCancelar.Enabled = FALSE
        RETURN
      END IF

    LOOP

    LblInfo.Text = "Inactivo"
    Message.Info("Descarga finalizada")

  END
```

En el evento Read() leemos una línea y la añadimos al contenido previo de TxtLicencia, más un retorno de carro para separar cada línea.

Evento Kill() y la propiedad Value

Además del método *Kill()*, la clase *Process* emite un evento Kill() cuando un proceso ha finalizado, sea de forma normal o por un error. La sintaxis de dicho evento es:

```
PUBLIC SUB Kill()
  ...
END
```

Hasta ahora, en nuestro gestor de descarga hemos esperado en un bucle a que el proceso acabe, pero podemos crear una estructura más elegante, más adaptada a la programación orientada a objetos, valiéndonos de dicho evento: en lugar de esperar en un bucle, aguardaremos tranquilamente y sin hacer absolutamente nada en nuestro programa principal hasta recibir el evento Kill(), bien sea porque la descarga ha finalizado, bien sea porque el usuario ha decidido cancelar el proceso:

```
PRIVATE hCancel AS Boolean
PRIVATE hProc AS Process
```

```
PUBLIC SUB BtnCancelar_Click()

  hCancel = TRUE
  hProc.Kill()

END

PUBLIC SUB Process_Error(Err AS String)

  DIM sCad AS String[]

  Err = Trim(Err)
  sCad = Split(Err, " ")
  LblInfo.Text = sCad[sCad.Count - 1]

END

PUBLIC SUB Process_Kill()

  IF hCancel = TRUE THEN
    Message.Warning("Proceso cancelado")
  ELSE
    Message.Info("Descarga finalizada")
  END IF

  LblInfo.Text = "Inactivo"
  BtnCancelar.Enabled = FALSE

END

PUBLIC SUB BtnDescarga_Click()

  DIM sCad AS String
```

```
    TxtLicencia.Text = ""
    hCancel = FALSE
    BtnCancelar.Enabled = TRUE
    hProc = EXEC ["curl", "http://www.gnu.org/licenses/
    gpl.txt", "-#"] FOR READ

END
```

Hemos cambiado la declaración de **hProc** de la función **BtnDescarga_Click()** al inicio de nuestro formulario, para que sea accesible también desde el evento **BtnCancelar_Click()**. Ahora lanzamos el proceso y ya no esperamos en un bucle. Si el usuario decide cancelar la descarga, matamos el proceso en el mismo evento **BtnCancelar_Click()**.

Tanto si el programa termina normalmente como debido a una cancelación, aprovechamos el evento **Process_Kill()** para informar al usuario y devolver la interfaz a su estado inicial (con el botón **BtnCancelar** deshabilitado, y la etiqueta marcando **Inactivo**).

El resultado final: menos gasto de recursos (el programa principal no ha de ejecutar el bucle constantemente), menos líneas de código y mejor estructuración de todo el proceso.

Por otro lado, la mayor parte de los programas de consola (y gráficos) devuelven al sistema un código de error cuando finalizan. La costumbre es que se devuelva un cero si todo fue bien y otro valor cuando algo falló.

Desde la consola podemos hacer una prueba ejecutando en una ventana de terminal estas dos órdenes consecutivas:

```
$ ls -l /dev
$ echo $?
```

Al ejecutar **echo $?** obtendremos un cero. La shell del sistema almacenó el último código de error disponible en la variable **$?** y, a continuación, lo hemos mostrado en pantalla: todo fue bien. Ahora ejecutamos la siguiente orden:

```
$ ls -l /fichero/que/no/existe
$ echo $?
```

En esta ocasión obtendremos un seis, un código de error que en el caso del comando **ls** implica que la ruta no existía (siempre y cuando no haya, por alguna extraña razón que desconozcamos, un fichero en nuestro sistema cuya ruta sea **/fichero/que/no/existe**).

Si *curl* no puede acceder a la URL, por cualquier circunstancia, devolverá un valor distinto de 0. Nosotros podemos leer en nuestro código ese valor, mediante la propiedad *Value*, e informar al usuario del error. A continuación, modificamos el código del programa anterior (página 142 en adelante), de forma que el evento *Kill* quede así:

```
PUBLIC SUB Process_Kill()

  IF LAST.Value<>0 THEN
    Message.Error ("Error en la descarga")
  ELSE
     IF hCancel = TRUE THEN
       Message.Warning("Proceso cancelado")
     ELSE
       Message.Info("Descarga finalizada")
     END IF
  END IF

  LblInfo.Text = "Inactivo"
  BtnCancelar.Enabled = FALSE

END
```

Probemos el programa desconectando el módem de Internet, desenchufando el cable de red o deshabilitando la red de nuestro sistema. Observaremos que ahora podemos determinar el éxito o fallo de la descarga, además de su cancelación.

Redirección de la entrada estandar, el uso de CLOSE

Los programas, además de emitir información a través de *stdin* y *stderr*, la pueden recibir a través de la consola, según el usuario teclea órdenes. Esta recepción se realiza mediante la entrada estándar o *stdin*, y si tenemos conocimientos de C sabremos que podemos utilizarla con funciones como *scanf()* o *getchar()*. Con Gambas, podemos emplear el flag *FOR WRITE* para indicar al intérprete que estamos interesados en escribir datos para el proceso hijo. Una vez lanzado el proceso de este modo, podemos usar las funciones normales de escritura de archivos con el descriptor del proceso (*PRINT*, *WRITE*).

Supongamos un programa con un área de texto (*TextArea*) llamada **TxtTexto**, en la que escribimos cualquier cosa, y deseamos conocer el número de líneas (separadas con retornos de carro), que hemos escrito. Podemos, para ello, usar el comando **wc** (significa *Word Counter* y no otra cosa), con el parámetro **-l** (número de líneas). En nuestro ejemplo, llamaremos al programa, escribiremos el contenido de **TxtTexto** al proceso, y recibiremos el resultado por la entrada estándar. Puesto que vamos a leer y escribir, podemos combinar los flags **READ** y **WRITE**. Para ello, creamos un proyecto gráfico con un botón **BtnContar**, con el texto **Contar**, y un *TextArea* llamado **TxtTexto**, con el texto en blanco inicialmente para que después escribamos un texto.

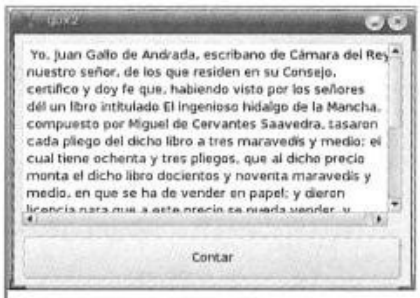

Figura 4. Proyecto con botón **Contar** y área de texto **TxtTexto**.

```
PRIVATE hProc AS Process

PUBLIC SUB Process_Read()

  DIM sCad AS String
```

```
    LINE INPUT #hProc, sCad
    Message("El texto tiene : " & sCad & " líneas")

  END

  PUBLIC SUB BtnContar_Click()

    hProc = EXEC ["wc", "-l"] FOR READ WRITE
    PRINT #hProc, TxtTexto.Text
    CLOSE #hProc

  END
```

n la función BtnContar_Click(), ejecutamos el programa indicando al intérprete ue deseamos acceso de lectura y escritura al proceso, escribimos en éste el conteido de la caja de texto y, a continuación, ejecutamos CLOSE sobre el proceso.

eguidamente, ejecutamos wc -l desde una terminal de línea de comandos. bservaremos que podemos ir escribiendo en el proceso todo lo que queramos. Así, cleamos varias líneas y después pulsamos a la vez las teclas **Control + D**.

s entonces cuando nos devuelve el número de líneas que hemos escrito. Al manar el carácter especial CTRL+D, estamos indicando al proceso que hemos cerrao el flujo de datos y, por tanto, procede que él compute lo recibido y devuelva el sultado.

uando en el programa Gambas ejecutamos CLOSE sobre un descriptor de fiche, el resultado es similar: se cierra la redirección entre la salida estándar del procehijo y nuestro proceso principal, con lo cual el primero queda informado de que recibido todos los datos.

resto del programa es trivial: en el evento READ recibimos la cadena que contiee el número de líneas y lo mostramos al usuario en un mensaje.

Notas finales sobre el objeto Process

El objeto *Process* tiene un método *Signal()* usado por el propio depurador de Gambas. Permite enviar señales al proceso en curso, pero su uso no es aconsejado, ya que interfiere con el resto del código del intérprete.

El objeto dispone también de una propiedad *Id*, que es un handle o descriptor del archivo. Si conocemos C o C++, se trata del identificador de proceso o PID del programa hijo que se obtiene tras una llamada a *fork()*, que es el modo de crear nuevos procesos en sistemas UNIX. Este valor puede servir para buscar el proceso, por ejemplo, ejecutando *ps -le*, variar su propiedad con *nice*, o utilizarlo junto con posteriores llamadas a funciones de C (veremos en otro capítulo cómo hacerlo desde Gambas) para controlar dicho proceso.

Los eventos generados al trabajar con procesos son estáticos. Habremos observado que en los sucesivos ejemplos se ha indicado **Process_Read** y **Process_Kill**. Si tenemos varios procesos en ejecución, siempre podemos diferenciar qué proceso ha generado el evento mediante la palabra clave **LAST** y actuar en consecuencia:


```
PRIVATE hProc1 AS Process
PRIVATE hProc2 AS Process
...
PUBLIC SUB Process_Read()
  IF LAST=hProc1 THEN ...

END
```

4. 4 SHELL

SHELL es similar a *EXEC*, pero en este caso lanza la shell o intérprete de comandos del sistema, habitualmente *BASH*, en las distribuciones GNU/Linux y le pasa el comando que le indiquemos. Esto permite, por ejemplo, dar órdenes al sistema tales como *export* o *cd*, que son parte de *BASH* y no comandos independientes.

También da vía libre al uso de tuberías | y operadores de redirección como > o *2>* entre otras características de la shell. La sintaxis de **SHELL** es la siguiente:

```
[Variable = ] SHELL Command [ WAIT ] [ FOR ( READ |
WRITE | READ WRITE )
```

Es muy similar a la de *EXEC*, pero en este caso **Command** es una cadena de texto y no una matriz de cadenas, ya que este valor se pasa directamente a la shell del sistema que, dependiendo de sus características, interpretará los espacios y otros símbolos especiales de un modo u otro. Queda pues, por tanto, como trabajo para el programador, añadir el formateo adecuado para que la shell interprete la orden correctamente.