Introducción a los scripts con Bash

Bash es la *shell de línea de comandos* creada para el sistema operativo GNU, y utilizada de forma predeterminada desde hace muchos años en los sistemas Linux y macOS. Su manual oficial *online* puede consultarse aquí, aunque cualquier sistema Linux tendrá una copia local que puede consultarse ejecutando `man bash`. Para ver una lista de los *comandos internos* que incluye, podemos ejecutar `help`.

La shell y sus guiones

Si partimos de la idea de que una *shell* de línea de comandos como **Bash** no es más que un programa del sistema operativo que espera nuestras órdenes (comandos) para cumplirlas (ejecutarlas) no debería extrañarnos el concepto de *shell script* o "*guión de la shell*". Teniendo en cuenta que este tipo de *shell* interpreta "palabras" (*cadenas de texto*) como nuestras órdenes, en lugar de irlas tecleando y ejecutando (al pulsar la tecla *Enter*) una a una, podríamos guardar en un archivo de texto una secuencia de estas órdenes que resulte útil para llevar a cabo alguna tarea concreta para poder volver a ejecutarla en cualquier momento. Para ello solo tendríamos que lanzar una nueva *shell* indicando de algún modo cuál es el "guión", la "lista de órdenes" que debe ejecutar.

Supongamos que el contenido del archivo de texto **lista_comandos.txt** es el siguiente:

```
date
who
free -h
```

La forma más simple de ejecución de uno de estos *scripts* o *guiones de la shell* sería por tanto teclear la siguiente línea de comandos:

```
$ bash lista_comandos.txt
```

A la *shell* abierta (indicada por el caracter **$**) le hemos pedido que ejecute el programa `bash` (otra *shell*) y que le pase a esta el archivo de texto `lista_comandos.txt` para que procese su contenido. El resultado sería la ejecución de las órdenes de texto que contiene dicho archivo, en el orden indicado: `date` (muestra fecha y hora actuales), `who` (muestra lista de usuarios conectados) y `free` (muestra la cantidad de memoria libre) . En este caso, la secuencia de esos tres comandos podría ser útil a algún usuario o administrador de un sistema Linux para hacerse una idea del estado del mismo.

Un par de mejoras que tienen que ver con el nombre del archivo utilizado, conteniendo la lista de órdenes a utilizar, podrían ser las siguientes:

1. Utilizar un nombre más descriptivo de la utilidad del *script*, en lugar de **lista_comandos** llamarlo por ejemplo, **estado_sistema**.
2. Utilizar otra extensión de archivo (aunque en Linux las extensiones solo las tiene en cuenta el sistema en entornos gráficos) más concreta, en lugar de **.txt** (texto) **.sh** (órdenes para la *shell*)

Ahora tendríamos un archivo llamado **estado_sistema.sh** que guardaríamos junto a otros *scripts* sin peligro de confusión, y que podríamos ejecutar cuando nos haga falta de igual modo:

```
$ bash estado_sistema.sh
```

Si necesitamos añadir alguna orden al *script* tan solo tendremos que editarlo con nuestro editor de texto favorito (desde la terminal lo más sencillo y estándar sería usar **nano**). Por ejemplo, podríamos añadir al final el comando `df -h` que lista el espacio libre en las distintas unidades de disco, con lo que el contenido completo quedaría así:

```
date
who
free -h
df -h
```

Haciendo de una lista de comandos un nuevo comando

Junto con los alias y las funciones de la shell, los *scripts* permiten al usuario/administrador personalizar o incluso ampliar las opciones disponibles a la hora de realizar sus tareas cotidianas en el sistema. Típicamente, un *script* se usará para automatizar una serie de tareas repetitivas o para crear un nuevo comando, a base de combinar comandos existentes, que le permita llevar a cabo una tarea muy concreta para la que no existía ningún comando apropiado. En ambos casos, pero sobre todo en el segundo, en que estamos creando un nuevo comando, es útil poder ejecutarlo como un comando más, simplemente escribiendo su nombre en la *shell* (y pulsando *Enter*). Para ello hay que indicar de algún modo al sistema operativo que ese archivo de texto donde hemos guardado una lista de órdenes, es un programa. Esto lo conseguiremos dándole permisos de ejecución al archivo:

`$ chmod +x estado_sistema.sh`

Al hacer eso, ya podremos ejecutarlo directamente sin tener que ejecutar explícitamente una nueva *shell* antes del nombre del archivo. Sin embargo no será tan fácil como teclear lo siguiente y pulsar *Enter*:

`$ estado_sistema.sh`

Debido a un mecanismo de seguridad de la *shell*, para poder ejecutar un archivo ejecutable del directorio actual tenemos que indicar **explícitamente** la ruta hasta el mismo, sea la *ruta absoluta* (desde el directorio raíz **/**) o la *ruta relativa* (desde el directorio actual), más habitual:

`$ ./estado_sistema.sh`

De todos modos, es habitual almacenar los distintos *scripts* en ciertos directorios concretos, bien del usuario o del sistema, que permitirán la ejecución directa sin tener que indicar explícitamente su *ruta*.

Directorios para nuestros scripts (y otros programas)

Si un usuario crea un subdirectorio llamado **bin** bajo su directorio personal (la *ruta completa* quedaría **/home/usuario/bin**) y copia allí sus *scripts*, podrá ejecutarlos directamente sin importar cuál sea el directorio actual en la *shell* que esté usando. Si copiamos nuestro archivo ejecutable **estado_sistema.sh** a este subdirectorio **bin** (si no existe lo creamos ejecutando `mkdir bin`), la próxima vez que iniciemos sesión ya podremos ejecutarlo directamente sin más que teclear su nombre (y pulsar *Enter*):

`$ estado_sistema.sh`

En un sistema Linux multiusuario (todos los son, pero aquí nos referimos a si va a usarse por más de un usuario) puede interesarnos almacenar todos los *scripts* y programas de los distintos usuarios en el directorio del sistema **/usr/local/bin**, que está pensado para esos programas "*locales*", personalizados en nuestro sistema, que podrán ejecutar todos los usuarios.

En cualquiera de estos casos, es habitual renombrar el *script* para que tenga el aspecto de un comando o programa más del sistema, quitando la coletilla "**.sh**", extensión que en realidad solo nos servía para distinguirlo mejor de otros archivos cuando lo teníamos almacenado en nuestro directorio personal. También podríamos cambiar los guiones bajos por guiones normales, más habituales en los nombres de comandos y programas. Tras renombrarlo convenientemente, la forma de invocarlo desde la *shell* sería:

```
$ estado-sistema
```

Antes de instalar uno de estos *scripts* de nuestra propia cosecha en el sistema, se recomienda comprobar antes que no existe ningún comando o programa instalado con ese mismo nombre. Para ello, antes de darle su nombre definitivo probaríamos a ejecutarlo directamente. Si la *shell* no encuentra ningún programa con ese nombre querrá decir que podemos usarlo sin problemas para nombrar nuestro *script*.

Variables para almacenar datos

La *shell* Bash, al igual que otros programas de su tipo, permite el uso de lo que se denominan *variables de shell*. Esto no son más que unos "*contenedores*" que crearemos en memoria, cada uno con su nombre, y donde podremos guardar datos para posteriormente recuperarlos usando su nombre. Este tipo de recursos es muy útil a la hora de convertir a las listas de comandos vistas hasta ahora en auténticos programas.

Para crear una nueva variable simplemente usaremos el operador de asignación, representado por el caracter "**=**":

```
$ mivariable=cualquierdato
```

Si la variable ya existe y volvemos a asignarle otro valor, se sobreescribirá el valor anterior con el nuevo.

Para acceder al contenido actual de una variable usaremos el caracter **$**, y podemos usar el comando **echo** para imprimirlo en pantalla:

```
$ echo $mivariable
```

Toda *shell* cuenta, cuando se ejecuta, con una serie de *variables de shell* ya definidas, que almacenan una serie de datos útiles para los programas que la *shell* vaya ejecutando. Se dice que estas variables definen cierto "entorno de ejecución" para ellos, por lo que se les llama *variables de entorno*. Estas variables están estandarizadas, y para distinguirlas de las *variables de shell* normales se escriben en mayúsculas. Un par de ejemplos pueden ser la variable USER y la variable SHELL, que almacenan el nombre del usuario actual y la *shell* que está ejecutando, respectivamente. Para ver sus valores usariamos de nuevo el comando `echo` y el caracter **$** justo antes del nombre de la variable:

```
$ echo $USER
```

Hay una *variable de entorno* especialmente importante, ya que contiene la lista de directorios del sistema donde la *shell* buscará los archivos ejecutables (comandos, *scripts* u otros programas) para su ejecución. Se trata de PATH, cuyo valor podremos consultar de igual manera:

```
$ echo $PATH
```

Un ejemplo de su contenido podría ser algo así:

```
/home/usuario/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

La *shell* buscará, en cada uno de los directorios enumerados (separados por el caracter "**:**") el comando o programa a ejecutar hasta encontrarlo, en cuyo caso se ejecutará, o hasta agotar la lista, en cuyo caso producirá en pantalla un error si no se encontrara se dará un mensaje de error del estilo:

`asdfg: no se encontró la orden`

Incorporando un par de *variables de entorno* al *script* de ejemplo, quedaría así:

```
echo "Usuario: $USER Shell: $SHELL"
date
who
free -h
df -h
```

Algunas variables especiales

Existen unas variables predefinidas que resultan especialmente útiles dentro de un *script*, llamadas **$1**, **$2**, **$3**, etc. que permiten recibir los distintos argumentos que se hayan pasado en línea de comandos tras el nombre del *script*, en el orden indicado por el número que da nombre a la variable.

Como ejemplo podemos imaginar un *script* llamado `actualiza-css` que procesa archivos **.html** para cambiar la hoja de estilo referenciada dentro del documento. El *script* se ejecutaría pasándole en línea de comandos el nombre del archivo **.html** a procesar, por ejemplo:

`$ actualiza-css acercade.html`

Dentro del *script* usuariamos la variable `$1` para acceder a este argumento pasado en la ejecución, en este caso el nombre del archivo que habrá que leer y procesar con los comandos necesarios.

Si necesitamos procesar en nuestro *script* un número indeterminado de argumentos pasados en su ejecución, por ejemplo una lista de archivos a procesar (podrían ser el resultado de usar *caracteres comodín* en la *shell* al estilo de ***.html**) podemos usar la variable especial **$@** para procesar dicha lista en un *bucle* **for** similar al del apartado **Estructuras de control II** de más abajo.

Otra variable predefinida que puede sernos útil dentro y fuera de un *script* es **$?** y contiene el valor (numérico) correspondiente al estado de finalización del último programa o comando ejecutado. Por ejemplo, si en una línea del *script* se ejecuta un comando, y justo a continuación se quiere decidir qué comandos ejecutar en base al éxito o no del comando anterior, podría usarse el valor de **$?** como condición en una estructura **if-then-else** de las que se ven más abajo en el apartado **Estructuras de control I**.

Shell predeterminada

Si no se indica lo contrario, la shell que se usará para interpretar y ejecutar los distintos comandos de un *script* será la predeterminada del sistema, que en la mayoría de sistemas Linux es Bash. Si queremos indicar en nuestro *script* que debe ser interpretado por una *shell* concreta, se usa un comentario especial al principio del mismo llamado *shebang*, que tendría este aspecto:

`#!/bin/bash`

Este simplemente indica la *ruta* hasta el ejecutable de la shell que debe usarse para procesar las líneas del *script*. Esto puede ser necesario porque existen otras *shells* (p. ej. Bourne Shell, Zsh, etc.) y cada una tiene unas características que permiten programarlas de forma distinta, y por tanto incompatible con las demás. Para asegurarnos de que se usa la *shell* adecuada cuando nuestro *script* se comparta y acabe ejecutándose en otro sistema tipo Unix/Linux, donde no sabemos *a priori* cuál sera la *shell* predeterminada, usariamos el *shebang*.

Funcionalidades de la shell que facilitan su programación

Hay algunas funcionalidades de toda *shell de línea de comandos*, y Bash no es una excepción, que la hacen más util, más potente, tanto en su uso interactivo a la hora de ejecutar comandos y programas, como a la hora de programarla mediante el uso de *scripts*.

Las **tuberías** de comandos consisten en combinar múltiples comandos, de forma que la *salida* (la información que muestra) de uno se canaliza hacia la entrada del siguiente. El símbolo utilizado es "**|**", llamado *pipe* o "tubería", y cuando se escribe entre los nombres de comandos o programas en una sola línea, la *shell* (Bash en nuestro caso) interpreta que debe conectarlos para formar esa tubería:

```
$ cat /etc/passwd | grep operador | cut -d: -f7
```

En el ejemplo anterior, el comando `cat` lee el contenido del archivo de texto `/etc/passwd` que contiene la información sobre las cuentas de usuario, pero en lugar de volcarlo a la pantalla, el uso de la tubería hace que todo ese texto se le pase al siguiente comando, `grep` que se utiliza para *filtrar* líneas de texto, en este caso aquellas en que aparezca la palabra "operador" (un supuesto nombre de usuario). La línea resultante, en lugar de aparecer por pantalla, vuelve a "entubarse" con otro comando, `cut`, especializado en *filtrar* columnas, en este caso la séptima (*field 7*) usando como *delimitador* de las mismas el caracter ":".

Este mecanismo de *conexión de programas*, permite combinar comandos y programas existentes, como si fueran *piezas de lego*, para realizar nuevas tareas, y resolver problemas concretos. Además de usar estas *tuberías* en la línea de comandos, si la guardamos en un archivo para usos posteriores tendremos la base de un típico *script*.

Una alternativa que nos ofrece una *shell* como Bash es la de desviar los datos de un comando, en vez de a otro comando como en el caso de las *tuberías*, a un archivo. A esto se le llama **redirección de la salida estándar** y para usarla se utiliza el símbolo "**>**" situado entre el nombre del programa y el del archivo de destino:

```
$ mount > discos_montados.txt
```

Si el archivo ya existe, se sobreescribirá su contenido con el nuevo generado por el comando. Si no deseamos esto, sino que preferimos que se añada al final, podemos duplicar el símbolo "**>**" de la siguiente forma:

```
$ mount >> discos_montados.txt
```

Un archivo especial que suele usarse en *scripts* para deshacernos de los datos de salida generados por un comando es `/dev/null` conocido como el *sumidero de bits*, porque todos los datos que se le envían, se pierden.

También podría desviarse el contenido de un archivo hacia la entrada de datos de un comando utilizando el símbolo opuesto "**<**" (ver ejemplo de *bucle while* más abajo). A esto se le llama **redirección de la entrada estándar**.

Otra posibilidad muy útil que nos ofrece Bash es la llamada *sustitución de comandos*, donde el resultado de la ejecución de un comando (el texto que genera) ocupa el lugar del comando mismo. Esto se le indica a la *shell* situando el comando en cuestión dentro de los símbolos **$( )** de la siguiente forma:

```
$ ls -l $(which netcat)
```

Primero la *shell* ejecuta el comando dentro de los paréntesis, en este caso `which netcat` que devuelve la ruta hasta el ejecutable del programa *netcat*, típicamente `/bin/netcat`. A continuación ejecuta el comando `ls -l` para listar el archivo obtenido en el paso anterior, con lo que sería como ejecutar en la misma línea dos comandos, pero con un orden concreto (primero los que estén entre paréntesis) y respetando la posición de cada uno en la línea de comandos. Otro ejemplo muy útil en *scripts*:

```
$ escaner=$(which nmap)
```

Se guardará en la variable *escaner* la ruta hasta el ejecutable del famoso *escáner de puertos* **nmap**. Esto permitiría en un *script* saber si dicho programa está instalado o no comprobando si la variable tiene algún valor o está vacía.

Estructuras de control I: condicionales

Aunque una secuencia de comandos que realiza cierta tarea, guardada en un archivo de texto para su reutilización, ya puede llamarse *script*, la potencia de estos empieza a desplegarse cuando se utilizan ciertas características de la *shell* que permiten un mayor control sobre *qué hace* el *script* y *en qué situaciones* lo hace. Esto es lo que se llama en programación *estructuras de control*, y vamos a empezar por las que permiten seleccionar qué comandos o instrucciones se ejecutan, en función de cierta condición:

```
if condición
then
    comando1
else
    comando2
fi
```

La *condición* que decidirá que comandos se ejecutan vendrá dada por el resultado de la ejecución de un comando. Todos los comandos acaban con un resultado de éxito (*código de salida* **0**) o de fallo (*código de salida* distinto a 0, normalmente **1**). Un ejemplo con el comando de diagnóstico de red `ping`:

```
if ping -c3 192.168.1.1
then
    echo "El router responde"
else
    echo "El router no responde"
fi
```

Esto puede aplicarse a cualquier comando, aunque hay algunos especializados para poder realizar comparaciones entre *cadenas de texto* o *números*, o consultar el *estado de un archivo* usándolos entre corchetes (caracteres "**[**" y "**]**"):

```
if [ "$texto1 = "$texto2" ]
then
    echo "Las cadenas de texto son iguales"
else
    echo "Las cadenas de texto son diferentes"
fi
```

```
if [ $num_lineas -gt 1000 ]
then
    echo "El archivo tiene más de 1000 líneas"
else
    echo "El archivo tiene 1000 líneas o menos"
fi
```

```
if [ -r /etc/samba/smb.conf ]
then
    echo "Existe el archivo de configuración de Samba y puede leerse"
else
    echo "No existe el archivo de configuración de Samba o no puede leerse"
fi
```

Para casos en que haya que elegir entre más posibles situaciones, por ejemplo entre un conjunto mayor de valores de una variable, podríamos usar la estructura de selección múltiple **case**.

Por otro lado, si queremos probar estas estructuras multilínea directamente en la línea de comandos o usarlas en un *script* con la instrucción `then` en la misma línea que la *condición*, podemos usar como separador el caracter "**;**" así: `if mount /dev/sdb1 /mnt; then echo "Unidad montada"; fi`

Estructuras de control II: bucles

Otra estructura básica para poder controlar la ejecución de comandos en un *script* es el llamado *bucle*, un bloque de código que se ejecutará repetidas veces en función de ciertas condiciones.

Uno de los *bucles* más útiles es el llamado **for**, que nos permite recorrer una lista de *objetos*, por ejemplo archivos, y ejecutar un bloque de comandos con cada uno de ellos:

```
for fichero in *
do
    file $fichero
done
```

En el ejemplo anterior, para cada archivo del directorio actual se ejecutaría el comando `file` que mostrará información sobre el tipo de archivo de que se trata.

Otra estructura de *bucle* muy usada es la llamada **while**, donde se repetirá la ejecución de un bloque de comandos mientras se cumpla cierta *condición*, y cuando esta deje de cumplirse, pasarán a ejecutarse los comandos posteriores al bucle. Un ejemplo con el comando `ping`:

```
while ping -c3 192.168.1.11 > /dev/null
do
    sleep 60
done
echo -e "Subject: Alerta\n\nFallo servidor" | ssmtp admin@empresa.com
```

Quizá haya que aclarar algunos detalles del ejemplo anterior:

- Se ha redirigido la salida de datos (por defecto a pantalla) del comando `ping` a **/dev/null** para deshacernos de ella, ya que solo interesa si acaba o no con éxito para seguir o no dentro del *bucle*.
- Dentro del *bucle* tan solo se ejecuta el comando `sleep 60` que sirve para "*dormir*" una cantidad de tiempo, normalmente en segundos, sin hacer nada.
- Al salir del *bucle* (cuando falle el comando `ping`) se ejecutará el siguiente comando, en este caso una tubería donde se "*entuba*" un escueto mensaje al programa de e-mail `ssmtp` (debe estar instalado y correctamente configurado en el sistema) para enviar una alerta al supuesto administrador de la empresa. La opción **-e** del comando `echo` permite intercalar en el texto caracteres especiales como el usado "**\n**", que representa una *nueva línea*, en este caso necesaria para formatear correctamente el mensaje.

Una variante muy útil de un *bucle* **while** consiste en ir leyendo, usando el comando `read`, las líneas de un archivo de texto (redireccionado a la *entrada estándar* del `while`), y para cada una ejecutar una serie de comandos:

```
contador=0
while read linea
do
    echo $linea
    contador=$((contador+1))
done < lista_usuarios.txt
echo "Total usuarios: $contador"
```

En el caso anterior, mientras el comando `read` tenga éxito a la hora de leer en la variable `linea` una nueva línea del archivo `lista_usuarios.txt`, se ejecutará el comando `echo` para mostrar por pantalla el valor actual de la *variable*, y además se incrementará en una unidad el valor de la variable **contador** usando los símbolos "**$(( ))**", reservados por Bash para realizar operaciones aritméticas, en nuestro caso una simple *suma*.

Si se quiere construir un bucle en que se repitan ciertos comandos *hasta* que se cumpla cierta condición, se utilizaría la estructura **until** en lugar de la *while*.

Shell scripts como "cinta plástica"

En los apartados anteriores se ha podido ver que una *shell* tipo Bash cuenta con una serie de funcionalidades que la hacen "*programable*" al estilo de cualquier lenguaje de programación estructurada (junto a las estructuras de *selección* e *iteración* pueden usarse también *subrutinas* llamadas funciones de la shell). Prueba de ello es que ha sido utilizada para crear aplicaciones de línea de comandos bastante potentes como el gestor de descargas Plowshare o instaladores de conocidas aplicaciones como JDownloader en su versión para Linux. Sin embargo, estas funcionalidades suelen aplicarse más a la *programación de sistemas* ocasional o la *automatización de tareas* rutinarias por parte de administradores y usuarios avanzados, casi siempre de una forma más "*artesanal*", pero también más rápida, de lo que se haría con lenguajes "serios" como C o Java. Esta forma de aplicación a la solución de problemas suele compararse con el uso de *cinta plástica* para construir o ensamblar una nueva herramienta hecha a medida a partir de una serie de herramientas existentes (comandos y utilidades). Por tanto, el potencial de un *script* va a estar claramente limitado por la colección de herramientas disponibles, listas para combinarlas y ensamblarlas con nuestra *cinta plástica*. En un típico sistema Linux como Ubuntu, podemos tener disponibles unos 1000 o 2000 comandos externos y utilidades (según se trate de un entorno de servidor o de escritorio), una cantidad nada despreciable de posibilidades. Cualquier programa de terminal disponible

en el sistema, lo vamos a poder ejecutar y controlar desde nuestros *scripts*, de la misma manera que desde un lenguaje de programación tradicional pueden usarse cientos de funciones y/o clases ya programadas disponibles en sus bibliotecas estándares. Desde este punto de vista, la desventaja más importante (rendimiento aparte) de los *shell scripts*, quizá la que los mantienen en el terreno de la "*programación artesanal*" y fuera de los modernos dominios de la "*ingeniería del software*", tiene que ver con la heterogeneidad de los componentes: las diferencias a la hora de usar los distintos programas y la consiguiente dificultad para integrarlos de forma sistemática y estandarizada.

Si es texto hay un camino

Bash, como cualquier *shell de línea de comandos* del mundo Unix, se diseñó para trabajar con comandos, programas y archivos basados en **texto**, por lo que puede utilizarse sin dudarlo para resolver todo tipo de problemas donde haya que manipular texto en cualquiera de sus formas. Además de las funcionalidades incorporadas en la propia *shell* para ello, podemos recurrir a decenas o incluso cientos de programas que se han creado a lo largo de décadas para generar, filtrar o procesar texto. Algunos imprescindibles y disponibles en cualquier Linux:

- grep - Seleccionar las líneas de texto que cumplan ciertos patrones.
- cut - Seleccionar secciones de cada línea de texto usando ciertos criterios.
- sort - Ordenar las distintas líneas de texto en función de diversos criterios.
- sed - Potente editor *no interactivo* capaz de transformar flujos de texto "al vuelo".
- awk - Intérprete con su propio lenguaje para escánear y procesar texto, normalmente en formato tabular y pudiendo hacer cálculos con las distintas columnas.
- Expresiones regulares - No es un programa, sino una tecnología consistente en definir secuencias especiales de caracteres que permiten buscar patrones de texto complejos, y que pueden usarse para aumentar la potencia de `grep`, `sed` y `awk`.

¿Multimedia y texto?

El hecho de que los *scripts* de Bash destaquen a la hora de procesar texto, no quiere decir que no se pueda, usando los programas y utilidades de línea de comandos adecuadas, manipular otros contenidos más "modernos", incluso **multimedia**:

- poppler-utils - Paquete de utilidades, preinstalado en algunos escritorios Linux, para la manipulación de documentos PDF.
- ImageMagick - Colección de herramientas especializadas en procesar y convertir imágenes.
- FFmpeg - Herramienta especializada en la grabación, conversión y *streaming* de audio y vídeo.
- SoX - La "*navaja suiza*" del procesamiento de audio, desde conversiones básicas hasta síntesis de sonidos y procesamiento digital de señales.

La internet de los scripts

La programación de Bash y su ámbito de aplicación no queda confinada a un ordenador local, sino que mediante las utilidades adecuadas, y no faltan en el típico repertorio Linux, puede extenderse a nuestra red local o incluso a los servidores de internet, sobre todo si usan *protocolos* basados en **texto**. Una pequeña muestra:

- wget - Utilidad de línea de comandos disponible en cualquier sistema Linux y que puede utilizarse para descargar archivos usando los protocolos típicos de la WWW (HTTP, HTTPS, FTP, etc.), desde un simple URL a toda una web para su navegación *offline*.

- curl - Utilidad de línea de comandos sucesora de `wget` en diversos ámbitos debido a que soporta los protocolos de la web y muchos otros (SMTP, POP3, SMB, etc.) para transferencia de archivos de modo no interactivo. Además puede encontrarse preinstalada en macOS además de en Linux.
- w3m - Sencillo navegador web basado en texto que interpreta el código HTML de los documentos web y lo formatea como texto plano, por lo que puede utilizarse, en su modo *no interactivo* (opción **-dump**) para tareas básicas de web scraping o "*raspado de datos de la web*".
- youtube-dl - Utilidad para la descarga de contenidos de la plataforma YouTube; no suele estar preinstalado en Linux (aunque sí disponible en los repositorios de las principales *distros*) pero se incluye por su utilidad actual y como ejemplo de utilidades de línea de comandos que siguen apareciendo para resolver todo tipo de problemas y necesidades de la era de Internet.

Escapando del terminal

Aunque el ámbito de aplicación de Bash suele estar limitado a las *terminales de comandos*, tanto en sistemas Linux de escritorio como de servidor, en el primer caso podemos hacer uso de las utilidades zenity o kdialog, dependiendo de nuestro entorno de escritorio, para comunicar nuestros *scripts interactivos* con el usuario mediante ventanas, los típicos "*cuadros de diálogo*" para entrada o salida de datos. En el caso de *scripts no interactivos* aun podrían usarse para enviar *notificaciones* al escritorio, algo bastante útil para mensajes de estado, alertas, etc.

Ejecución en el inicio de sesión

Cualquier usuario de un sistema Linux puede tener la necesidad de ejecutar automáticamente alguno de sus *scripts* cuando inicie sesión en el sistema. Para ello puede usarse el fichero `.profile` que reside en el directorio personal de cada usuario y se usa para que estos personalicen su *perfil* de usuario en el sistema, es decir sus respectivas cuentas de usuario inicializando *variables de entorno*, ejecutando comandos, etc.

Por ejemplo, si tenemos un *script* llamado `prevision-tiempo` que nos muestra el pronóstico meteorológico en pantalla y queremos iniciarlo al iniciar sesión, tan solo tendremos que añadirlo a nuestro archivo `.profile`:

```
# Fichero .profile del usuario
...
prevision-tiempo
```

Existe una versión *global* de dicho archivo, `/etc/profile` que solo el *superusuario* (administrador) del sistema puede cambiar y que afectaría al inicio de sesión de todos los usuarios.

Ejecución en el inicio del sistema

Si lo que necesitamos es ejecutar un `script` cuando arranque el sistema operativo, independientemente de si los usuarios inician sesión o no, el archivo de configuración a utilizar es `/etc/rc.local`, donde el *superusuario* podrá añadir la ruta hasta el `script` a ejecutar, respetando al final la línea `exit 0`:

```
# Fichero /etc/rc.local del sistema
...
nuestro-script
exit 0
```

Hay que tener en cuenta que cuando se invoque desde **rc.local** un *script* o programa, este se ejecutará en nombre del usuario **root**, y por tanto con privilegios máximos en el sistema. Si no es lo que deseamos, podemos usar el comando **su** para "suplantar" a otro usuario a la hora de ejecutar el *script* de la siguiente forma:

`su usuario -c nuestro-script`

Ejecutar con cierta frecuencia

Es algo habitual en cualquier sistema operativo moderno querer ejecutar ciertos *scripts* o programas periodicamente, cada cierto tiempo: una vez al día, a la semana, al mes, etc. Para ello existe un servicio del sistema llamado **cron** que lee ciertos archivos de configuración y directorios para llevar a cabo estas ejecuciones periódicas.

Cada usuario tiene una *tabla para cron* propia que puede editar con el comando `crontab -e`. La sintaxis de cada línea donde se definen las distintas tareas a ejecutar es algo así:

```
#m h dom mon dow command
15 *  *   *   *   script-del-usuario
```

El *script* del ejemplo anterior se ejecutaría el *minuto* **15** de cada *hora*, *día del mes* y *día de la semana*. Para ver el manual electrónico instalado en el sistema sobre los detalles de uso de estas *tablas para cron* podemos ejecutar el comando `man 5 crontab`.

Al igual que pasaba con el archivo `.profile`, existe una versión global de esta *tabla para cron* en el fichero `/etc/crontab` que solo el *superusuario* podrá editar para programar la ejecución periódica de ciertos *scripts* u otros programas. Para ello debera editarlo directamente con su editor preferido, como `nano` o `vi`. La sintaxis es similar, excepto por una columna adicional donde se indica en nombre de qué usuario del sistema se ejecutará el programa cuando llegue su hora, por defecto **root**.

Por último, si la frecuencia de ejecución del *script* o programa no requiere de una hora precisa, podemos usar los directorios del sistema existentes bajo `/etc` para copiar en ellos nuestro `script` (o una referencia al mismo) y, dependiendo del directorio concreto, conseguir una frecuencia u otra de ejecución. Los directorios disponibles son `/etc/cron.hourly` para ejecutarse cada hora, `/etc/cron.daily` para una vez al día, `/etc/cron.weekly` para una frecuencia semanal y `/etc/cron.monthly` para una mensual.

Para profundizar...

- El shell Bash, de Fernando López para MacProgramadores. Lo mejor gratis en español.
- The Linux Command Line, por William Shotts. Uno de los mejores en Inglés y además gratis.
- CommandLineFu.com es un repositorio de líneas de comandos para hacer (casi) cualquier cosa.