INTRODUCCION A

SHELL SCRIPT

1. INTRODUCCION.

Shell que significa en Castellano “concha” es el interprete de comandos del sistema. Es una interfaz de texto de altas prestaciones, que sirve fundamentalmente para tres cosas: administrar el sistema operativo, lanzar aplicaciones (e interactuar con ellas) y como entorno de programación. Gnu/Linux se administra editando ficheros configuración. Como norma general, se encuentran en: **/etc**, dentro de direcciones específicos para cada aplicación. Por ejemplo, **Lilo** (Linux Loader) se configura editando el fichero: **/etc/lilo/lilo.conf**. Los programas, se ejecutan escribiendo el nombre del ejecutable, si este se encuentra en el path (ruta por defecto para los mismos, normalmente: **/usr/bin**) o escribiendo el nombre del ejecutable precedido por: **./**, desde el directorio donde se encuentren. Todo esto, es bien conocido por cualquier usuario de **Shell**. No tan conocidas son sus capacidades como entorno de programación. Los programas de **Shell** no necesitan compilarse. La **Shell** los interpreta linea a linea. Se les suele conocer como **Shells Scripts** y pueden ser desde sencillas ordenes hasta complejas series de instrucciones para el arranque del propio sistema operativo. En general, tienen una sintaxis bastante clara y suponen un buen punto de partida para dar los primeros pasos en el mundo de la programación.

Yo no soy ningún experto programador. De hecho, estoy aprendiendo ahora mismo. Es un mundo apasionante, pero un poco oscuro a veces. Entonces, si no soy un especialista en el tema, ¿Como me atrevo a escribir sobre ello? Bueno, tengo mis motivos. Verán, me considero un buen comunicador, y creo que mi forma de contar las cosas puede ser útil a los demás. Ello me anima a preparar tutoriales como estos, y a publicarlos en Internet.

2. ¿Como se edita un script de Shell?

El enfoque de este trabajo es práctico. En lugar de mostrar el funcionamiento de cada comando, vamos a ver ejercicios concretos que sean apropiados para la administración de nuestro sistema. Los **scripts** de **Shell** son extremadamente útiles. Es buena idea escribir aquellas necesidades que tengamos y luego editar **scripts** que realicen este trabajo por nosotros. A estas alturas, ya es hora de preguntarse que es exactamente un **script**. Es un archivo de texto, que contiene una serie de comandos para **shell**, que el sistema ejecuta ordenadamente, de arriba abajo. Para editarlos, tan solo hace falta un editor de textos, como **Emacs**, o **Vi**. Se guardan con extensión: **.sh** y se ejecutan desde la **Shell** mediante el comando: **sh nombre de script.sh**. Los **scripts**, se comportan de igual manera que los comandos de la **shell**.

Bueno, ya va siendo hora de que pasemos a la práctica. El primer paso para trabajar con una **Shell** es ejecutar una **shell**. Esto que parece una perogrullada tiene su razón de ser. En algunas distribuciones de **Gnu/Linux** muy enfocadas al usuario finál, la **shell** está bastante escondida. Normalmente, se llama: Konsole, terminal, terminal de X, o algo similar. Se suele incluir en el menú k de **Kde**, en el apartado sistema. Otra opción es utilizar una consola virtual. Mediante: **Ctrl alt f1**, o **f2**, o **f3** hasta **f6**, podemos utilizar entornos **shell**, sin interfaz gráfica. Es importante saber, que los programas basados en el entorno **X**, lógicamente no funcionan. La **Shell** más utilizada en **Gnu/Linux** es **Bash**, aunque existen otras, como **ksh** o **C Shell**. Este tutorial está enfocado a **Bash**.

3. PASAMOS A LA ACCION.

Supongamos que evitamos el arranque automático del escritorio gráfico. Esto es bastante simple. Solamente hay que cambiar de nombre o comentar el fichero: **gdm**, **kdm** o **xdm**. Esto depende de nuestro sistema. **Kdm** está vinculado a **Kde**, **Gdm** a **Gnome**, y **Xdm** a cualquiera de ellos. Los programas que inician el sistema operativo son también scripts. Se encuentran en el directorio: **/etc/init.d.** Hecho esto, el sistema arranca en modo de texto. para lanzar el servidor gráfico:

startx ruta al escritorio -- : número de nivel de ejecución

Aparte de **Kde** y **Gnome**, son bastante utilizados **windowmaker**, **icewm** y **enlightement**, entre otros. Si una vez ejecutado el escritorio, abandonamos la sesión, volvemos a la linea de comandos. Desde aquí, podemos preparar nuestros **scripts**. Veamos algunas ideas. Podríamos hacer uno que mostrase un menú para ejecutar el escritorio que queramos. Veamos, lo podemos llamar **escritorio.sh**. Lo editamos, este es el contenido:

```
clear
echo “MENU”
echo “====”
echo “1. Iniciar kde.”
echo “2. Iniciar Gnome.”
echo “3. Iniciar Windowmaker.”
echo “4. Iniciar enlightement.”

echo “Elige opción”
read entrada1
case $entrada1 in
1)
startx kde
;;
2)
startx gnome
;;
3)
startx windowmaker
;;
4)
startx enlightement
;;
esac
```

Y ahora veamos como funciona. El comando **clear** (primera linea), limpia la pantalla. El comando “**echo**” muestra lo que escribamos a continuación entre comillas dobles. Las linea dos a la ocho dan lugar a un menú, que sirve para escoger el escritorio que queramos lanzar. Para ello, el usuario tiene que escribir un numero, entre 1 y 4. A continuación, vemos el comando “**read**”. Este, lee la respuesta del usuario, y la guarda como la variable: **entrada1**. Ahora se utiliza la estructura “**case**”. Esta, permite realizar de una forma bastante sencilla, varios **if** anidados. La estructura **if fi**, consiste en evaluar condiciones: “si tal cosa es de tal forma, entonces haz esto. En caso contrario esto otro”. Mediante “**case**” nos ahorramos bastante texto. La sintaxis es la que se puede ver en el ejemplo. Empieza por **case** “**$variable**” **in**. Luego, en la siguiente linea: **1)** Es para referirse al valor: **1** , que en este caso es “**iniciar Kde**”. El resto son las entradas para los demás escritorios. Cada una de ellas termina obligatoriamente con los caracteres: **;;** Finalmente, para cerrar la estructura se escribe: **esac,** que no es otra cosa sino **case**, escrito al revés.

No es muy complicado. De esta forma, se pueden hacer muchas cosas. Por ejemplo, un script que nos sirva para apagar la máquina, rebotarla, etc. De hecho, si solemos utilizar mucho la consola, unos cuantos scripts de este tipo, nos evitará el estar escribiendo siempre lo mismo. Veamos otro ejercicio diferente. Es una calculadora. Es muy simple, solo sabe sumar, restar, multiplicar y dividir. Pero también es muy fácil de escribir y de utilizar. Ilustra bien el uso de variables.

```
echo “soy Gnu/Cal”
echo “Tu me dices lo que quieres calcular y yo te doy el resultado”
echo “Introduce el primer valor”
read valor1
echo “ Introduce el operador. Puedes escoger entre: + - * /”
read operador
echo “ Introduce el segundo valor”
read valor2
echo “El resultado es:”
sleep 2
expr $valor1 $operador $valor2
sleep 1
echo “gracias por su tiempo!”
```

Las tres primeras lineas son, por así decirlo, de presentación. En la cuarta linea hay un **read**, llamado **valor1**. Es una variable, que va a introducir desde el teclado el usuario. Después, es necesario saber que operación se quiere realizar (suma, resta, multiplicación o división). Para ello hay un nuevo **read**, llamado “**operador**” para que el usuario pueda escoger. Después, tenemos el segundo valor de la operación llamado “**valor2**”. El comando “**sleep**” lo único que hace es esperar un poco de tiempo para que de la sensación de que el programa esta pensando poco antes de dar el resultado. La operación matemática propiamente dicha se realiza en la linea 11. El comando es: **expr**. Como puede verse, opera directamente con los valores que hayan tomado las tres variables implicadas. Este tipo de estructura, es muy fácil de comprender (por eso forma parte del ejercicio) pero no es completamente funcional. Como veremos más tarde, hay otras formas de programar una calculadora. El mayor problema que presenta esta, es la operación multiplicación. El símbolo: * no es válido, porque está reservado para el sistema operativo.

Veamos ahora otro ejemplo. Se llama: **trivial.sh**, y es el típico juego de preguntas y respuestas. Solo tiene dos preguntas, pero es suficiente para ver como funciona. Está construido, sobre la estructura **if fi**. El programa conoce la respuesta correcta y evalúa la respuesta del usuario. Si acierta una pregunta se lo dice y pasa con la siguiente pregunta. Si acierta todas las preguntas le da el titulo de “magister del universo”.

```
#!/bin/bash
clear
echo "Bienvenidos a GnuTrivial."
sleep 2
echo "Soy el típico juego de preguntas y respuestas."
sleep 2
echo "Si aciertas todas las preguntas, te concedere el titulo de magister del universo."
echo "¿Como se llamaba el ultimo emperador Romano de occidente, claudio, teodosio o romulo."
read respuesta1
if test $respuesta1 = romulo
then
echo "Respuesta correcta."
else
echo "Lo siento, la respuesta correcta es: romulo."
fi
sleep 2
echo "Pasemos a la siguiente pregunta. ¿Qué célebre filosofo Griego tuvó por discipulo a Alejandro Magno, platón, aristoteles o zenon?"
read respuesta2
if test $respuesta2 = aristoteles
```

```
then
echo "respuesta correcta."
else
echo "Lo siento, la respuesta correcta es: Aristoteles."
fi
if test $respuesta1 = romulo
  test $respuesta2 = aristoteles
then
echo "Eres un pequeño magister del universo."
fi
```

Como puede apreciarse, la estructura **if fi**, y la estructura **case** hacen cosas bastante similares. Llevan a cabo una acción en función de un valor del usuario. En general, **case** se considera una evolución de **if fi**, ya que permite el mismo resultado, escribiendo menos código. Sin embargo, el uso de **if fi** es perfectamente correcto. La evaluación de las condiciones se realiza mediante el comando **test**. Este, comprueba si la variable es igual al valor que le hayamos indicado. Por supuesto, **test** puede hacer muchas más cosas, como comparar si un número es mayor o menor que otro, etc. Al finál del script, se evalúan dos variables a la vez. Es necesario escribirlas en dos lineas diferentes.

A continuación, vamos a hacer algo diferente. Se trata de una especie de minibase de datos para nuestra colección particular de películas. Se llama **videobase.sh**. Lo primero que tenemos que hacer es un script que genere un fichero de texto que contenga nombres de directores, nombres de peliculas, año de producción y productora. Cada linea son los datos de una película. Viene a ser lo que se llama un registro. Nuestro segundo script, es un cliente de este fichero de texto, con capacidad para realizar busquedas según varios criterios. Veamos el código:

```
#/bin/sh
clear
echo "VIDEOBASE"
echo "========="
echo "ESCOGER CRITERIO DE BUSQUEDA"
echo "1.- Busqueda por director."
echo "2.- Busqueda por productora."
echo "3.- Busqueda por titulo."
echo "4.- Busqueda por año."
echo "5.- Salir."

read criterio

case $criterio in
1)
echo "¿Cual es el nombre del director?"
read director
grep $director /home/antonio/base_pelis/base2.txt
read pausa
/home/antonio/bash-script/pelibus.sh
;;
2)
echo "¿Cual es el nombre de la productora?"
read productora
grep $productora /home/antonio/base_pelis/base2.txt
read pausa
/home/antonio/bash-script/pelibus.sh
;;
3)
echo "¿Cual es el titulo de la pelicula?"
read titulo
```

```
grep $titulo /home/antonio/base_pelis/base2.txt
read pausa
/home/antonio/bash-script/pelibus.sh
;;
4)
echo "En que año se produjo la pelicula?"
read ano
grep $ano /home/antonio/base_pelis/base2.txt
read pausa
/home/antonio/bash-script/pelibus.sh
;;
5)
;;
esac
```

La estructura es **case**. A estas alturas, ya estamos familiarizados con ella. Introduce un par de elementos nuevos. El primero de ellos, es que desde este script se llama a un script de nombre: **pelibus.sh**, es decir se llama a si mismo. Esto es, para que cuando el usuario pulse la tecla "intro" se vuelva a repetir el programa. Produce la sensación de volver atrás. La variable **pausa**, detiene el programa hasta que el usuario pulse alguna tecla. Por último, la opción **salir** se logra escribiendo: **5)**, sin añadir ningún tipo de orden. El sistema interpreta que el script ha terminado y nos devuelve a la linea de comandos. Las busquedas se llevan a cabo sobre un fichero de texto simple llamado: **base2.txt**, el comando para filtrar la información es: **grep**. Veamos ahora el fichero de texto que contiene los datos:

```
heroes fuera de orbita 1999 tim allen dreamworks
simone 2002 al pacino new line produccions
nivel 13 1999 roland emmerich columbia
el tercer hombre 1947 carol reed circulo
juan nadie 1945 frank capra circulo
```

Como se ha comentado antes, este fichero se crea y edita mediante otro script. Se llama "**pelin.sh**". Veamos su contenido:

```
#/bin/sh
clear
echo "VIDEOBASE"
echo " ========"
echo "NUEVO REGISTRO EN CURSO"
sleep 2
echo "-DIRECTOR:"
read director
sleep 1
echo "-PELICULA:"
read pelicula
sleep 1
echo "-AÑO DE PRODUCCION:"
read ano
sleep 1
echo "-PRODUCTORA:"
read productora
echo "$director $pelicula $ano $productora" >>/home/antonio/base_pelis/base2.txt
```

La parte interesante es la última linea. Un **echo** envia por la salida las cuatro variables en uso. Mediante: **>>** la salida no va a la pantalla (salida estandar) sino a un fichero llamado: **base2.txt**. En caso de no existir, lo crea. Si existe añade los datos nuevos. Si utilizamos: **>** sustituiríamos la información antigua por la nueva.

De la misma manera, podríamos escribir **scripts** para organizar una biblioteca, recibos bancarios o cualquier otra cosa. También se podría ampliar este programa con más opciones. Por ejemplo, una entrada para un código alfanumérico la dotaría de verdadera funcionalidad. Mediante este código podemos identificar los objetos que queramos clasificar y localizarlos físicamente. No he incluido más opciones porque no aportan nada más, desde el punto de vista didáctico. Una vez que se comprende la técnica, cualquiera puede escribir sus propios programas.

Continuemos adelante. Al principio hemos visto una calculadora, capaz de sumar, restar y dividir correctamente, pero que fallaba con la multiplicación. Veamos ahora el código de una calculadora capaz de operar correctamente también con la multiplicación.

```
echo ##########################
echo #       HOLA! SOY GNUCAL  V2        #
echo ##########################
echo
echo Introduce el primer valor:
read valor1
echo "Introduce operador, puede elegir suma +, resta -, multiplicacion * y division /."
read ope
echo Introduce segundo valor:
read valor2
resultado=`expr $valor1 "$ope" $valor2`
echo Resultado=$resultado
```

La estructura ha cambiado un poco. La idea es la misma, pero está mejor construida. La última linea muestra en pantalla la variable: **resultado**. Esta, esta asignada en el propio programa y se corresponde con: **`expr $valor1 “$ope” $valor2`**. El problema de la multiplicación se resuelve fácilmente incluyendo la variable **operador** entre comillas dobles. La estructura de este **script** es interesante, ya que se puede aplicar a otro tipo de operaciones simplemente cambiando el valor de la variable **resultado**.

4. EXPRESIONES REGULARES.

Las expresiones regulares son patrones que afectan a una cadena de caracteres. Son muy útiles para seleccionar con gran precisión elementos de un fichero. Se utilizan mucho en los scripts, y son la parte más difícil de entender. Veamos un ejemplo. Lo primero, creamos un fichero de texto llamado **expre** con el siguiente contenido:

cosas	**casa**	**libros**	**animal**	**verdugo**
LLUvia	**lanaa**	**madre**	**hermano**	**nido**
bicho	**limo**	**asno**	**vision**	**alma**
libro	**cosas**	**tipo**	**falso**	**oso**
bicha	**corbata**	**talon**	**barco**	**tigre**
ult				

Ahora vamos utilizar **grep** para buscar la cadena cosas dentro del fichero expre. Veamos, escribimos: **grep cosas expre**. Obtenemos:

libro	**cosas**	**tipo**	**falso**	**oso**
cosas	**casa**	**libros**	**animal**	**verdugo**

Como el fichero **expre** tiene cinco columnas, cuando **grep** selecciona la cadena cosas, nos muestra las filas completas en las que se encuentre. Ahora vamos a hacer uso de expresiones regulares para lograr selecciones más precisas. **“^cosas”** hace que **grep** busque la linea que

comience por cosas. Esto es consecuencia del metacaracter: **^**. Las expresiones regulares son precisamente caracteres especiales (llamados metacaracteres) que influyen de una manera particular en la cadena de texto que referencian. Hagamos la prueba. Escribimos: **grep "^cosas" expre**. El resultado es:

cosas casa libros animal verdugo

Es importante no escribir ningún carácter previo a la primera columna de palabras al editar el fichero **expre**. No es correcto utilizar la barra espaciadora ni tabuladores. **grep** los tomará por caracteres y en este caso, como ninguna linea empieza por "cosas" no mostrará resultado alguno. Veamos otro ejemplo. Supongamos que queremos filtrar todos los subdirectorios del directorio: **/usr**. Para ello escribimos:

ls -la /usr | grep ^d

Obtenemos:

```
drwxr-xr-x   13 root root   368 Sep 15 00:56 .
drwxr-xr-x   21 root root   488 Sep 24 14:03 ..
drwxrwxrwx    8 root root   192 Aug 13 17:15 X11R6
drwxrwxrwx    2 root root 54696 Sep 15 20:13 bin
drwxr-xr-x    2 root root  1384 Sep 15 01:27 games
drwxrwxrwx    5 root root   120 Aug 13 17:08 i586-suse-linux
drwxrwxrwx   97 root root 12864 Sep  8 00:02 include
drwxrwxrwx  123 root root 49648 Sep 15 20:13 lib
drwxrwxrwx   12 root root   304 Sep  7 00:57 local
drwxr-xr-x    4 root root    96 Sep 15 00:56 man
drwxrwxrwx    2 root root  9984 Sep  8 00:03 sbin
drwxr-xr-x  133 root root  3464 Sep 15 20:15 share
drwxrwxrwx    5 root root   160 Aug 26 00:09 src
```

El comando **grep** ha seleccionado únicamente aquellos ficheros que comienzan por la letra **d**. La primera columna, comenzando por la izquierda son los permisos. El primer carácter indica el tipo de fichero. d significa directorio. Bien, esta forma de proceder va a ser nuestra estrategia para administrar el sistema. filtraremos los ficheros de configuración del sistema en busca de la información que necesitemos para después hacer algo con ella. Continuemos. Si escribimos:

grep "^...$" expre

Obtenemos:

ult

En este caso, hemos seleccionado solamente aquella linea que empiece por una palabra de tres caracteres. Si la expresión regular fuese: **^....$**, entonces seleccionariamos palabras de cuatro caracteres. Basta, con escribir tantos puntos como caracteres tenga la palabra que queramos seleccionar. Otro ejemplo:

grep "^os[oa]" expre

Obtenemos:

bicho	**limo**	**asno**	**vision**	**alma**
bicha	**corbata**	**talon**	**barco**	**tigre**

Grep ha seleccionado las lineas que contienen bicho o bicha. Veamos otras posibilidades:

grep "[A-Z] [A-Z]*" expre

Obtenemos:

LLUvia lanaa madre hermano nido

Lo que **grep** ha hecho ahora es seleccionar la única linea que empieza por consonantes. Ahora vamos a buscar las lineas que terminan por el carácter: e. Escribimos:

grep 'e$' expre

Obtenemos:

bicha corbata talon barco tigre

Por último, probemos a buscar aquella linea donde se repite el carácter: a. Escribimos:

grep “a\{2,\}” expre

Obtenemos:

LLUvia lanaa madre hermano nido

Como no se trata de probar todas y cada una de las expresiones, he preparado el siguiente listado. Es buena idea practicar utilizando cada una de las expresiones y ver el resultado.

PATRON	REPRESENTA
bicho	**La cadena bicho**
^bicho	**La cadena bicho al comienzo de una linea**
bicho$	**La cadena bicho al final de una linea**
^bicho$	**La cadena bicho formando una única linea**
bich[ao]	**Las cadenas bicha y bicho**
bi[^aeiou]o	**La tercera letra no es una vocal minúscula**
bi.o	**La tercera letra es cualquier carácter**
^....$	**Cualquier linea que contenga cuatro caracteres cualesquiera**
^\.	**Cualquier linea que comience por un punto**
^[^.]	**Cualquier linea que no comience por un punto**
bicho$	**bicho, bichos, bichoss, bichosss, etc**
“*bicho”*	**bicho con o sin comillas dobles**
[a-z] [a-z]*	**Una o más letras minúsculas**
[^0-9a-z]	**Cualquier carácter que no sea ni número ni letra mayúscula.**
[A-Za-z]	**Cualquier letra, sea mayúscula o minúscula.**
[Ax5]	**Cualquier carácter que sea A, x o 5**
bicho | bicha	**Una de las palabras bicho o bicha**
(s | arb)usto	**Las palabras susto o arbusto**
\<bi	**Cualquier palabra que comience por bi**
bi\>	**Cualquier palabra que termine en bi**
a\{2,\}	**Dos o más aes en una misma fila.**

Como se ve, el mundo de las expresiones regulares no es tan oscuro como parece al principio. Más adelante, intentaremos aplicar nuestros nuevos conocimientos a la edición de scripts. Ahora vamos a ver otros temas.

6. LOS FILTROS.

Los comandos que sirven para filtrar información se denominan filtros. Uno de los más conocidos es **grep**. Sin embargo, existen otros. Vamos a ver en primer lugar el filtro **cut**. Permite cortar columnas o campos de un fichero de texto y enviarlos a la salida estándar. La opción **-c** es para cortar columnas y **-f** para cortar campos. La opción **-d** se utiliza para indicar que carácter hace de separación entre campos, es decir el delimitador. Si no se indica nada, el delimitador es el tabulador. Para que **cut** sepa que campos o columnas tiene que cortar hay que indicarlo con una lista. Esta lista tiene tres formatos posibles:

1-2 Campos o columnas de 1 a 2.
1- Campo o columna 1 toda la linea.
1, 2 Campos o columnas 1 y 2.

Veamos unos ejemplos. Para ello editamos un fichero de texto llamado amigos con el siguiente contenido:

ANT:3680112
SEX:6789450
COM:3454327
VIM:4532278
DAO:5468903

Escribimos en la consola:

cut -c 1-3 amigos

Obtenemos:

ANT
SEX
COM
VIM
DAO

Es decir, hemos cortado las tres primeras letras de cada linea del fichero personas. Ahora vamos a intentar algo un poco más difícil. Vamos a cortar campos individuales. El carácter delimitador del fichero amigos, es **:**, Si escribimos:

cut -d ':' -f 1 amigos

Obtenemos:

ANT
SEX
COM
VIM
DAO

O sea el primer campo. Si escribimos:

cut -d ':' -f 2 amigos

Obtenemos el segundo campo:

3680112
6789450
3454327
4532278
5468903

Ahora vamos a ver un ejemplo más útil. Vamos a filtrar el fichero passwd, que se encuentra en /etc y vamos a seleccionar los usuarios que utilizan el interprete de ordenes bash. Lo primero para ver el contenido del fichero escribimos: **cat /etc/passwd**. Obtenemos:

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
mail:x:8:12:Mailer daemon:/var/spool/clientmqueue:/bin/false
news:x:9:13:News system:/etc/news:/bin/bash
uucp:x:10:14:Unix-to-Unix CoPy system:/etc/uucp:/bin/bash
games:x:12:100:Games account:/var/games:/bin/bash
man:x:13:62:Manual pages viewer:/var/cache/man:/bin/bash
at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
wwwrun:x:30:8:WWW daemon apache:/var/lib/wwwrun:/bin/false
ftp:x:40:49:FTP account:/srv/ftp:/bin/bash
gdm:x:50:15:Gnome Display Manager daemon:/var/lib/gdm:/bin/bash
postfix:x:51:51:Postfix Daemon:/var/spool/postfix:/bin/false
sshd:x:71:65:SSH daemon:/var/lib/sshd:/bin/false
ntp:x:74:65534:NTP daemon:/var/lib/ntp:/bin/false
vdr:x:100:33:Video Disk Recorder:/var/spool/video:/bin/false
nobody:x:65534:65533:nobody:/var/lib/nobody:/bin/bash
antonio:x:1001:0:antonio becerro martinez:/home/antonio:/bin/sh

Escribimos:

cat /etc/passwd | grep bash | cut -d ':' -f 1,7

Obtenemos:

root:/bin/bash
bin:/bin/bash
daemon:/bin/bash
lp:/bin/bash
news:/bin/bash
uucp:/bin/bash
games:/bin/bash
man:/bin/bash
at:/bin/bash
ftp:/bin/bash
gdm:/bin/bash
nobody:/bin/bash

Hemos utilizado, en primer lugar **cat** para pasar el fichero **passwd** por la salida estándar, depués mediante un pipeline (pipe o tubería) lo hemos filtrado con **grep** seleccionando solamente aquellas lineas que contienen la cadena bash. Finalmente, nos quedamos con los campos 1 y 7.

Dejamos **cut**, y pasamos al comando **tr.** Es un filtro que se emplea como traductor, generalmente para convertir de minúsculas a mayúsculas y viceversa. Lo primero editamos un fichero de texto de nombre gnu que tenga el siguiente contenido:

Gnu/linux es un sistema operativo
QUE PRESENTA UN GRAN FUTURO

Escribimos en la consola:

tr [A-Z] [a-z] < gnu

Obtenemos:

gnu/linux es un sistema operativo
que presenta un gran futuro

Hemos convertido todos los caracteres entre A y Z en sus homólogos entre a y z. Lógicamente, si queremos hacer lo contrario escribimos:

tr [a-z] [A-Z] < gnu

Obtenemos:

GNU/LINUX ES UN SISTEMA OPERATIVO
QUE PRESENTA UN GRAN FUTURO

Ahora vamos a ver como sustituir caracteres. Escribimos:

tr [A-Z] g < gnu

Obtenemos:

gnu/linux es un sistema operativo
ggg gggggggg gg gggg gggggg

Todas las mayúsculas han sido sutituidas por el carácter **g**. Naturalmente, podemos modificar el rango de caracteres implicados si en lugar de utilizar: **[A-Z]** utilizamos: **[A-M]**, es decir que solo se van a convertir en el carácter **g**, las mayúsculas entre la **A** y la **M**. Probemos:

tr [A-M] g < gnu

Obtenemos:

gnu/linux es un sistema operativo
QUg PRgSgNTg UN gRgN gUTURO

También podemos utilizar las técnicas de sustitución para eliminar determinados caracteres. Se utiliza la opción **-d**. Veamos:

tr -d [A-Z] < gnu

Obtenemos:

nu/linux es un sistema operativo

Como era de esperar, todas las consonantes han sido eliminadas. Conviene señalar, que **tr** no afecta para nada al fichero de entrada, que como recordaremos se llama gnu. Lo toma, lo procesa, y envía el resultado al monitor. El original no es modificado en ningún caso.

Por último, vamos a ver el filtro **tee**. Tiene como misión bifurcar la salida de una orden enviando los resultados simultaneamente a la salida estándar (monitor) y a un fichero. Su sintaxis es:

tee [-a] archivo(s)

La opción **-a** significa append o sea añadir. Con esta opción la salida a un fichero no sobreescribe la información sino que la añade al final del fichero de texto. Este filtro es útil en algunos casos.

7. BUCLES.

Aparte de lo que hemos visto hasta ahora, existen tres construcciones conocidas como iteraciones o bucles. Son **while**, **until** y **for**. Sirven para ejecutar ciclos de comandos si se cumple una condición. Empecemos en primer lugar por **while**. Su sintaxis es:

```
while condicion
do
        orden
done
```

while evalúa la condición. Si esta es correcta, entonces ejecuta el o los comandos especificados. Si no es correcta, salta a la siguiente orden que haya después de la palabra reservada done. Correcto, quiere decir que el código devuelto por la condición sea 0. Es conveniente reseñar que en este caso se vuelve a iterar, es decir que se repite la construcción **while**. Por ello nos referimos a este tipo de construcciones como iteraciones, sentencias de control de flujo o bucles. Veamos un ejemplo:

```
############################
#       PROGRAMA DE EJEMPLO        #
#   DE LA SENTENCIA DE CONTROL     #
#            DE FLUJO WHILE        #
############################
        a=42
        while [ $a -le 53 ]
        do
              echo Contador = $a
              a=`expr $a + 1`
        done
```

La variable **a** tiene el valor **42**. Si se cumple la condición: **[$a -le 53]**, se ejecuta el comando **echo Contador = $a**. Es decir, si a es menor o igual que **53** (cosa que es evidentemente cierta) muestra en la pantalla **Contador = 42**. Pero no termina ahí la cosa. Como la condición ha sido cumplida, **while** vuelve a iterar. Debajo del comando **echo** hay una linea para dar una valor nuevo a la variable a. Se trata de: **a=`expr $a + 1`**, que simplemente quiere decir que el valor de **a** sea incrementado en una unidad. Bueno, **while** hará precisamente esto hasta llegar a **53**. El programa terminará ya que cualquier número mayor no cumplirá la condición dada, y por que no existe ningún otro comando después de **done**.

La construcción **until** es muy similar a **while**. De hecho, comparte la misma sintaxis. La diferencia consiste en que el código de retorno de la condición debe ser falso (distinto de 0) para iterar. Si es verdadero saltará a el comando que vaya a continuación de **done**. Veamos un ejemplo:

```
################################
#            PROGRAMA DE EJEMPLO          #
#      DE LA SENTENCIA DE CONTROL DE      #
#                FLUJO UNTIL              #
################################

until [ $a = hal9000 ]
do
  echo "Bienvenidos a Programa1. Para continuar escribe contraseña"
  read a
done
  echo "contraseña correcta"
```

La condición es que la variable **a** sea igual a la cadena **hal9000**. En caso contrario, vuelve a iterar. Si el usuario escribe la cadena correcta, entonces la sentencia de control de flujo termina. En este caso, continua con el comando **echo “contraseña correcta”** ya que se encuentra después de **done**.

Finalmente, vamos a estudiar el bucle **for**. Su sintaxis es como sigue:

```
for variable in lista
do
   ordenes
done
```

Es diferente a **while** y **until**. En la construcción **for** no hay que cumplir ninguna condición. **for,**

simplemente utiliza como variable la primera cadena de una lista, ejecutando las ordenes que le indiquemos a continuación. En la siguiente iteración utilizará la siguiente cadena de la lista, y así sucesivamente. Veamos un ejemplo:

```
############################
#        PROGRAMA DE EJEMPLO       #
#   DE LA SENTENCIA DE CONTROL DE  #
#              FLUJO FOR                 #
############################

for a in antonio eva fernado joseba julio
do
   mail $a < texto.txt
done
```

El programa es bastante sencillo.Toma nombres de usuarios del sistema que se le proporcionan en una lista (**antonio eva fernando joseba julio**) y les envía un correo utilizando como contenido del mismo el fichero **texto.txt**.

Y esto es todo. Espero que todo esto le sirva para dar sus primeros pasos en el mundo de la programación, como a mí mismo me está sirviendo.

Antonio Becerro Martinez

becerrodelinux@yahoo.es

Alcobendas 2005