INTRODUCCIÓN A SHELL SCRIPT

Hamnlet G. Rivera Lagares
<hrivera@codigolibre.org>

Primera Edición
2 De Noviembre del 2007

FUNDACIÓN CÓDIGO LIBRE DOMINICANO **F C L D**

http://www.codigolibre.org

Índice general

Capítulo1

Introducción

1.1. Conceptos preliminares

Shell que en castellano significa concha, es el intérprete de comandos del sistema. Es una interfaz de texto de altas prestaciones, que sirve principalmente para tres cosas: administrar el sistema operativo, lanzar aplicaciones (e interactuar con ellas) y como entorno de programación. GNU/Linux se administra editando archivos de configuración. Como norma general, se encuentran en: **/etc** dentro de directrios específicos para cada aplicación. Los programas se ejecutan escribiendo el nombre del ejecutable, si este se encuetra en el **PATH** (ruta por defecto para los mísmos, normalmente: **/usr/bin**) o escribiendo el nombre del ejecutable precedido por: ./, desde el directorio donde se encuentren. Los programas del **Shell** no necesitan compilarse. El **Shell** los interpreta línea a línea. Se les suele conocer como **Shell Scripts** y pueden ser desde sencillas ordenes hasta complejas series de instrucciones para el arranque del propio sistema operativo. En general, tinen una sintaxis bastante clara y suponen un buen punto de partida para dar los primeros pasos en el mundo de la programción. Yo no soy ningún experto programador. De hecho, estoy aprendiendo ahora mismo. Es un mundo apasionante, pero un poco oscuro aveces. Entonces, si yo no soy un especialista en el tema, Cómo 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 un tutorial como éste, y a publicarlo.

1.2. Porqué la programación en el shell?

Incluso existiendo varias interfaces gráficas para GNU/Linux el shell sigue siendo una herramienta muy eficiente. El **shell** no es sólo una colección de comandos sino un lenguaje de programación.: El **shell** es muy útil para tareas de administración, puedes comprender rápidamente si tus ideas funcionan, lo que lo hace muy útil tanto para crear un prototipo como para crear pequeñas utilidades que realicen tareas relativamente simples, donde la eficiencia importa menos que la facilidad de configuración, mantenimiento y portabilidad.

Capítulo2

Qué es un shell-script

2.1. Definiciones

Un **script** es como se llama a un archivo o conjuto de archivos que tienen instrucciones (en nuestro caso comandos de bash), y que necesitan de un programa ayudante para ejecutarse (en nuestro caso el propio terminal bash será el programa ayudante).

Un **script** tiene cierta similutud con un programa, pero existen diferencias. Generalmente, los **scripts** son archivos que contienen en formato texto, los comandos o instrucciones que la aplicación ayudante ejecutará.
Aunque en principio esto puede resultar un tanto abstracto, vamos a verlo despacio y con ejemplos. Si no has programado nunca, en este capítulo encontrarás el aprendizanje del **shell-scripting** en bash como una sencilla introducción a la programción, que te ayudará a comprender después los fundamentos de otros lenguajes de programación. No te preocupes, porque lo vamos a explicar todo de forma sencilla y no te quedará ninguna duda. Esta lección te permitirá dar un salto enorme en el aprovechamiento de tu SO GNU/Linux. Algunas partes del SO que no pertenecen al kernel están escritas en **shell-scripts**.

2.2. Los shells del sistema UNIX

2.2.1. Los shells más conocidos

Bash	(Bourne Again Shell):Este shell es estándar con caracteristicas del shell C. Este shell es estándar de los sistemas GNU/Linux
Csh	El shell de C de Berkley
Jsh	El shell de trabajo, una extensión del shell estandar
Ksh	El shell Korn
Sh	El tradiocinal shell de Bourne
Tcsh	Versión mejorada de sch
Zsh	Vesión mejorada de ksh

Capítulo3

Programación del shell

Para escribir un script, necesitamos:

1. Escribir los comandos del shell en un archivo de texto.
2. Hacer el archivo ejecutable (chmod +x archivo).
3. Escribir el nombre del archivo en la líena de comandos.

Cabe hacer algunas observaciones, si queremos utilizar otro tipo de shell para que interprete los comandos, debemos colocar como primera línea en el archivo de texto el nombre del shell; por ejemplo si nuestros comandos serán interpretados por el shell ksh, entonces colocamos como primera línea: **#!/bin/ksh**

3.1. Sección de ejemplos

Ahora que has practicado con bash y con el editor vim, ya estás preparado para crear tu primer **script** bash. Si recordamos, un **script** era un archivo de texto plano con comandos; así que haz **vim miscript.sh** y pon en él lo siguiente (lo explicaremos determinadamente a continuación):

```
#!/bin/bash
# Esta línea será ignorada
# Esta también

echo “Hola”

echo “Soy un script de shell.”
```

3.1.1. Ejemplos detallados

Observemos la primera línea: #!/**bin**/**bash**. Esta línea es característica de todos los **scripts** GNU/Linux. Tras #! indicamos la ruta a la aplicación ayudante, la que interpretará los comandos del archivo. En nuestro caso es bash, así que ponemos ahí /**bin**/**bash**, que es la ruta hacia la aplicación bash. Las ultimas líneas son dos comandos, los que luego serán ejecutados por bash. Como sabemos, el comando **echo** saca por **stdout** o salida estandar (por defecto por pantalla) lo que le pasemos como argumento, en esta caso dos frases. Guardamos el script y salgamos de vim. Ahora lo ejecutaremos. La primera forma de hacerlo es como sigue:

$ bash miscript.sh

Hola

Soy un scprit del shell

Como vez, todos los comandos del scprit han sido ejecutados. Aunque la forma mas frecuente de ejecutar scripts es, darles priemro permisos de ejecución, como ya sabemos del capítulo de usuarios, grupos y permisos:

$chmod +x miscript.sh

$./miscript.sh

Hola

Soy un script del shell

Al darle permisos de ejcución, podemos ejecutar ahora nuestro **script** como si de programa normal se tratase. Esto es posible gracias a la primera línea del script, que hace que se llame inmediatamente a la aplicación ayudante para procesar los comandos.

Veamos un segundo ejemplo, llámelo **operación.sh**

```
#!/bin/bash
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"
```

Procedamos a ejecutarlo

$ chmod +x operación.sh

$ ./operación.sh

Las tres primeras líneas son por así decirlo, de presentación. En la cuarta línea hay un **read**, llamado valor1. Es una variable que el usuario va a introducir desde el teclado. Después, es necesario saber que operación se quiere realizar (suma, resta 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 con el comando **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 esta reservado para el sistema operativo.

Veamos ahora otro ejemplo, se llama **juego.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 título de magistrado del del universo.

```
#/bin/bash
clear
echo "Bienvenidos a GNU/Juego"
sleep 2
echo "Soy el típico juego de preguntas y respuestas"
sleep 2
echo "Si aciertas todas las preguntas te concederé le título de magistrado del universo"
echo "Cómo se llama el fundador del preyecto GNU: stallman, torvalds o MIT ?"
read respuesta1
if test $respuesta1 = stallman
then
echo "Respuesta correcta."
else
echo "Lo siento, la respuesta correcta es: Richard Stallman."
fi
sleep 2
echo "Pasemos a la siguiete pregunta. Qué célebre filosofo tuvo por discípulo a Alejandro Magno: platón, aristoteles o zenón?"
read respuesta2
if test $respuesta2 = aristoteles
then
echo "respuesta correcta."
else
echo "Lo siento, la respuesta correcta es Aristoteles."
fi
if test $respuesta1 = stallman
test $respuesta1 = stallman
then
echo "Eres un pequeño magistrado del universo."
```

Como puede apreciarse la estructura **if fi**, y la estructura **case** hacen cosas similares. Llevan a cabo una acción en función del valor del usuario. En general, **case** 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** (más adelante hablaremos de éste). Este, comprueba si la variable es igual la valor que le hayamos indicado. Por su puesto, **test** puede hacer muchas cosas más, como comparar si un número es mayor o menor que otro, etc. Al final del scprit, se evalúan dos variables a la vez. Es necesario escribirlas en dos líneas diferentes.

3.2. Depuración de shell scripts

La ayuda más sencilla para depurar un **script** es el comando **echo**. Lo puedes usar para imprimir variables específica en el lugar donde sospeches que está el fallo. Probablemente esto es lo que usan la mayoría de programadores de shell el 80 % de veces que intentan encontrar un error. La ventaja de la programción del shell es que no requiere ninguna recompilación e insertar una sentencia **echo** se hace rápidamente.

El shell tambien tiene un modo de depuración real. Si hay un error en tu **script "scriptconerror.sh"** entonces puedes depurarlo con:

sh -x scriptconerror.sh

La opción **-x** del comando sh lista los comandos y los argumentos que son ejecutados. Así puedes ver qué partes del **script** se han ejecutado cuando ocurre el error, es decir, ejecutará el **script** y mostrará todas las sentencias que se ejecutan con las variables y comodines ya expandidos.

El shell tambien tine un modo para comprobar errores de sintaxis sin ejecutar el programa. Se usa así:

sh -n tu_script.sh

La opción **-n** del comando sh hace que no se ejecute ningún comando, solo chequea la sintaxis. Si no retorna nada, entonces tu programa no tiene errores de sintaxis.

La opción **-e** en modo no interactivo, hace que si un comando falla se termine inmediatamente el script.

La opción **-v** imprime las líneas de entrada según son leídas.

Capítulo4

El comando test

El lenguaje **shell-scprit** muy frecuentemente ejecuta comandos externos del sistema operativo. Queremos explicar el comando **test** antes de explicar las sentencias condicionales de bash porque de esta forma podemos usar ejemplos más completos.

Test es un comando externo del sistema que devolverá un código de retorno 0 o 1 dependiendo del resultado de la expresión que siga a continuación. Se usa mucho y tiene dos formatos **test <expr>ó [expr]**. Ambas formas son equivalentes. Las expresiones de este típo que solo pueden valer TRUE o FALSE se denominan expresiones lógicas. **Test** retorna un valor igual a 0 si la expresión evaluada es verdadera y un número diferente de 0 si la expresión es falsa. **Test** permite evaluar cadenas, enteros y el estado de archivo del sistema UNIX.

4.1. Tabla de comprobación sobre ficheros con el comando test

Tabla de comprobación de ficheros

-r <fichero>	Verdadero si el archivo existe y es legible
-w <fichero>	Verdadero si el fichero existe y se puede escribir
-x <fichero>	Verdadero si el fichro existe y es ejcutable
-f <fichero>	Verdadero si el fichero existe y es de tipo regular
-d <fichero>	Verdadero si existe y es un directorio
-c <fichero>	Verdadero si existe y es dispositivo especial de caractes
-b <fichero>	Verdadero si existe y es un dispositivo especial de bloques
-p <fichero>	Verdadero si existe y es un pipe (fifo)
-u <fichero>	Verdadero si existe y tiene el bit set-user-UID
-s <fichero>	Verdadero si existe y tiene tamaño mayor que cero
-S <fichero>	Verdadero si existe y es un socket
-t [fd]	Verdadero si fd esta abierta en un terminal
-S <fichero>	Verdadero si existe y es un socket
-L <fichero>	Verdadero si existe y es un link simbolico
-e <fichero>	Verdadero si el fichero existe
-k <fichero>	Verdadero si el fichero tiene su “sticky” bit set

4.1.1. Tabla de comprobación de cadenas

Tabla de comprobación de cadenas

-z <cadena>	Verdadero si es una cadena vacía
-n <cadena>	Verdadero si es una cadena no vacía
<cadena1>= <cadena2>	Verdadero si cadena1 y cadena2 son idénticas
<cadena1>!= <cadena2>	Verdadero si cadena1 y cadena2 son distintas
<cadena1>	Verdadero si cadena1 no es una cadena nula

4.1.2. Tabla de operadores lógicos o de comprobación

Operadores de comprobación

! <expresión_lógica>	Operador negación
-a <expresión_lógica>	Operador binario AND
-o <expresión_logica>	Operador binario OR

Ejemplos:

```
$ test –r /etc/passwd ; echo $?
0
$ test –w /etc/passwd ; echo $?
1
$ test –x /etc/passwd ; echo $?
1
$ test –c /dev/null ; echo $?
0
$ test –r /etc/passwd –a –c /dev/null ; echo $?
0
$ test –w /etc/passwd –a –c /dev/null ; echo $?
1
$ test –r /etc/passwd –a –f /dev/null ; echo $?
1
$ [ –s /dev/null ] ; echo $?
1
$ [ ! –s /dev/null ] ; echo $?
0
$ [ "$$" = "zzzzzzzzzzzz" ] ; echo $?
1
$ [ 0 –lt $$ ] ; echo $?
0
$ [ 0 –lt $$ –a true ] ; echo $?
0
```

4.1.3. Tabla de comprobación de enteros

Relación de equivalencia

Escritura matemática	Relación de equivalencia	Significado matemático
$>$	-gt	Mayor que
$<$	-lt	Menor que
$=$	-eq	Igual que
$\neq$	-ne	Diferente de
$\leq$	-le	Menor o igual que
$\geq$	-ge	Mayor o igual que

4.2. El comando expr

Este comando adminte como parámetros los elementos de una expresión arimética **(ver man expr)**, pero hay que recordar que algunas operadores deben ser escapados por la barra invertida (backslash) "\".

Por ejemplo:
$ expr 11 \ ∗ 2

A diferencia de **bc** no podemos usar valores muy altos, ya que el resultado no sería correcto. Recuerde que **bc** trabaja con precisión arbitraria. El comando **expr** es algo ineficiente.

4.2.1. Expresiones aritméticas dentro de bash

El **shell** por defecto asume que todas las variables son de tipo cadena de caracteres. Para definir una variable de tipo numerico se usa **typeset -i**. Esto le añade un atributo a esa variable para que el shell realice una expresión distinta sobre esa variable. Para evaluar una expresión arimética asignándola a una variable usaremos **let**.

– +	Menos unário y Más unário
* /%	Multiplicación, División y Resto
+ –	Suma y Resta
<=, >=, <, >	Menor o igual, Mayor o igual, Menor, Mayor
== !=	Igualddad, Desigualdad

Ejemplo:

```
$ typeset –i j=7
$ typeset –i k
$ typeset –i m
$ echo $j
7
$ let j=j+3
$ echo $j
10
$ let j+=3
$ echo $j
13
$ let k=j %3
$ let m=j/3
$ echo '('$m '* 3 ) +'$k '='$j
+AMA
( 4 * 3 ) + 1 = 13
```

Puede que el operador % no le resulte familiar. Es el operador resto, también llamado módulo y es lo que sobra después de la división.

4.3. Operadores && y ||

Son los operadores AND y OR a nivel de **shell** y poseen circuito de evaluación corto. No tienen que ver con los operadores –a y –o, ya que eśtos eran interpretados por el comando **test**. Estos operadores && y || serán colocados separando comandos que lógicamente retornan siempre algun valor.

Por ejemplo:

'<comando1>&& <comando2>&& <comando3>'significa que debe cumplirse <comando1>y <comando2>y <comando3>para que se cumpla la expresión y '<comando1>|| <comando2>|| <comando3>'significa que debe cumplirse <comando1>o <comando2>o <comando3>para que se cumpla la expresión.

Si el resultado de '<comando1>&& <comando2>&& <comando3>'fuera 0 (TRUE) Significaría que los tres comandos han retornado 0 (TRUE). Si por el contrario el resultado hubiera sido distinto (FALSE) solo sabríamos que por lo menos uno de los comandos retornó FALSE. Supongamos que el primer comando retornó TRUE, en este caso el shell continurá ejecutando el segundo comando. Supongamos que el segundo comando retorna FALSE. En este momento el shell no continua ejecutando el tercer comando por que da igual, lo que retorne el resultado será FASLE. La razón es que se necesitaban todos true para un resultado TRUE.

Con el operador '||'(OR) pasa algo similar. Si el resultado de '<comando1>|| <comando2>|| <comando3>'fuera 1 (FALSE) significaría que los tres comandos han retornado 1 (FALSE). Si por el contrario el resultado hubiera sido distinto 0 (TRUE) solo sabríamos que por lo menos uno de los comandos retornó TRUE. Supongamos que el primer comando retornó FALSE. El shell deberá continuar ejecutando el segundo comando. Ahora supongamos que el segundo comando retorna TRUE, en este momento el shell no continua ejecutando el tercer comando porque da igual lo que retorne, el resultado será TRUE. Se necesitaban todos FALSE para un resultado FALSE.

Un comando normal como por ejemplo **grep o echo** que termina bien devuelve TRUE. Si hubiéramos redirigido la salida de grep o de echo habría sido FALSE indicando la existencia de error. En otras ocasiones FALSE no indicará error sino que el comando no tuvo éxito. Este último sería aplicado a grep, pero no sería aplicable al comando **echo**.

Veamos un ejemplo con grep:

```
$ echo hola | grep hola; echo $?
hola
0
$ echo hola | grep hola >/hh/hh/hh; echo $?
bash: /hh/hh/hh: No existe el fichero o el directorio
1
$ echo xxx | grep hola; echo $?
1
```

Cuando ejecutamos comandos con un pipe (|) el código de retorno es el del último comando ejecutado. En nuestro caso el primer código 1 se debe a la imposibilidad de generar la salida. El segundo código 1 se debe a que la cadena no fue encontrada. En los tres casos $? recoge el código retornado por grep.

4.3.1. Sentencia condicional if

Ya hemos visto una forma de hacerlo usando los operadores || y && del shell, pero existen formas que pueden resultar más versátiles y más legibles. El formato es el siguiente:

if condición
then lista_de_ordenes

elfi condición
then lista_de_ordenes
...
else condición

fi

En lugar de condición podríamos haber puesto lista de ordenes pero queremos resaltar que el código de retorno va a ser evaluado.
Todas las sentencias condicionales **if** empiezan con la palabra reservada **if** y terminan con la palabra reservada **fi**. Vamos a ilustrarlo con algunos ejemplos:

Esto siempre mostrará 123
if true
then echo ‘123‘
fi

Acabamos de utilizar una condición que siempre se cumplirá. Vamos a ver algo un poco más útil.
Si la variable FICH está definida y contine el nombre de un fichero con
#permiso de lectura se mandará a la impresora con lpr.

if test –r $FICH
then lpr $FICH
fi

También podemos programar una acción para cuando se cumpla una condición y otra para cuando no se cumpla.
#Si la variable modo contiene el valor lpr, imprimir el fichero $FICH, en caso contrario deberá sacarlo por pantalla.

```
if ["$modo" = "lpr" ]
then lpr $FICH
else cat $FICH
fi
```

El siguiente ejemplo edítelo con el nombre 'tipofichero'. Nos servirá para ilustrar el uso de **elif** y para repasar algunas de las opciones de **test**.

```
#tipofichero
FILE=$1
if test –b $FILE
then echo "$FILE Es un dispositivo de bloques"
elif test –c $FILE
then echo "$FILE Es un dispositivo especial de caracteres"
elif test –d $FILE
then echo "$FILE Es un directorio"
elif test –f $FILE
then echo "$FILE Es un fichero regular (ordinario)"
elif test –L $FILE
then echo "$FILE Es un Link simbólico"
elif test –p $FILE
then echo "$FILE Es un pipe con nombre"
elif test –S $FILE
then echo "$FILE Es un Socket (dispositivo de comunicaciones)"
elif test –e $FILE
then echo "$FILE Existe pero no sabemos que tipo de fichero es"
else echo "$FILE No existe o no es accesible"
fi
```

Otra utilidad del **if** es la posibilidad de realizar lo que se denomina **i**f anidados. De esta forma podríamos tener varias capas de **if–then–else–fi**. Como ejemplo mostraremos la siguiente construcción.

```
if [$condicion1 = "true"]
then
if [$condicion2 = "true"]
then
if [$condicion3 = "true"]
then
echo “Las condiciones 1,2 y 3 son ciertas”
else
echo “Solo son ciertas las condiciones 1 y 2”
fi
else
echo “La condición 1 es cierta, pero no la 2”
fi
else
echo “La condición 1 no es cierta”
fi
```

Capítulo5

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**.

5.1. Sentencia while

Su sintaxis es:

while condición
do
orden
done

while evalúa la condición. Si ésta 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 vuelva a iterar, es decir que se repita la construcción **while**. Por ello nos referimos a este tipo de construcciones como iteraciones, sentencias de control de flujo o bucles.

Veamos algunos ejemlos:

```
#!/bin/bash
a=10
while [ $a -le 20 ]
do
echo cont = $a
a=‘expr $a + 1‘
done
```

La variable a tiene un valor de **10**. Si se cumple la condición **[$a -le 20]**, se ejecuta el comando
cont = 10. Pero no termina ahí la cosa. Como la condición ha sido cumplida, **while** vuelve a iterar. Debajo del comando **echo** hay una línea para dar un 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 **20**. El programa terminará, ya que cualquier número mayor no cumplirá la condición dada, y porque no existe nigún otro comando después de **done**.

Lo mismo sucede en este otro ejemplo:

```
#Script que muestra la hora del sistema cada segundo durante un minuto
#!/bin/bash
cont=0
while [ $cont -lt 60 ]
do
cont=‘$cont + 1‘
sleep 1
date
done
```

5.2. Sentencia until

La construcción **until** es muy similar a **while**. De hecho, comparte la misma sitaxis. La diferencia consiste en que el código de retorno de la condición debe ser falso (distinto de cero) para iterar. Si es verdadero saltará a el comando que vaya a

continuacion de **done**.

Veamos un ejemplo:

```
until [ $password = codigolibre ]
do
echo "Bienvenidos a este programa. Para continuar escriba la contraseña"
read password
done
echo "contraseña correcta"
```

La condición es que la variable **password** sea igual a la cadena **condigolibre**. 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**.

5.3. Sentencia for

Vamos a estudiar el bucle **for**. Su sintaxis es como sigue:

```
for variable in lista
do
órdenes
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:

```
for usuario in antonio jorge dionisio jose
do
mail $usuario <mensaje.txt
done
```

El programa es bastante sencillo. Toma nombres de usuarios del sistema que se le proporcionan en una lista **(antonio jorge dionisio jose)** y les invía un correo utilizando como contenido del mismo fichero **mensaje.txt**.

5.4. break y continue

Existe una forma de controlar un bucle desde el interior del mismo. para eso podemos usar **break o continue**. Se pueden usar en cualquiera de los bucles que acabamos de ver **(while, until, for)**. La palabra reservada **break** provoca la salida de un bucle por el final. Si viene seguido de un número **n** saldrá de **n** niveles. No poner número equivale a poner el número 1. la Palabra reservada **continue** provoca un salto al comienzo del bucle pra continuar con siguiente iteración. Si viene seguida de un número **n** saldrá de **n** niveles.

Veamos algunos ejemplos:

Lláme al primero **break_contine.sh**

```
#!/bin/bash
#Definir la variable j como una variable tipo entero e inicializarla
#en cero,luego la incrementamos a cada iteración del bucle y si es
# menor que diez mostraremos el doble de su valor,
# en caso contrario salimos del bucle.

j=0
while true
do
j=‘expr $j + 1‘
if [ $j –eq 3 ]
then continue
fi
if [ $j –eq 4 ]
then continue
fi
if [ $j –lt 10 ]
then expr $j “ * ” 2
else break
fi
done
```

Pasamos ahora a ejecutarlo y obtendremos

$ ./break_continue.sh

2
4
10
12
14
16
18

```
#!/bin/bash
while echo "Por favor introduce un comando"
read respuesta
do
case “$respuesta” in ‘fin’)
break # no mas comandos
;;
“ ”)
continue # comando nulo
;;
*)
echo $respuesta # ejecuta el comando
;;
esac
done
```

5.5. La sentencia condicional case

La construcción es la siguiente:

```
read ELECCION
case $ELECCION in
a) programa1;;
b) programa2;;
c) programa3;;
*) echo "No eligió ninguna opción valida"
;;
esac
```

Hay que tener en cuenta algunas cosas respecto a éste tipo de construcción. Por ejemplo el mandato que le damos al principio **read** indica al **shell** que tiene que leer lo que se ingrese a continuación y lo guarde en una variable que se llamana ELECCION. Esto también será útil para el uso de otras construcciones ya que **read** no es propidad exclisiva de la construcción **esac**, sino que pertenece al mismo shell. Como se ve se le indica que siel valor que la variable contiene es igual a alguna de las mostradas debajo, se ejecute determinado programa. **(case ELECCION in)**. La elección se debe terminar con un paréntesis para que se cierren las posibilidades. Podríamos poner más posibilidades para cada elección; lo único que hay recordar es cerrar con un paréntsis. El punto y coma nos marca el final de un bloque, por lo que podríamos agregar otro comando y se cerrará con punto y coma doble el último. El asterisco del final nos indica qué se hará en caso de que no coincida lo ingresado con ninguana de las posibilidades.

Ilustremos algunos ejemplos usando la sentencia **case** y construyendo algunos menues.

Este es nuestro primer ejemplo, llámelo calc.sh

```
#!/bin/bash
#Script disenado para simular una caluladora que multiplica la tabla
de un numero introducido desde el 1 hasta el 12
max=12
while true
clear
do
#MENU DE OPCIONES#
echo -n “

**********MENU DE OPCIONES**********
1. Ver la tabla de un numero introducido
2. Salir del programa
Elija su opcion... ”

read op
case $op in
1) echo “ ” echo -n “Introduzca el numero que desea multiplicar ”
read num
echo “ Imprimiendo los resultados”
echo “ ”
i=0
while [ $i -lt $max ]
do
i=‘expr $i + 1‘
result=‘expr $num “*” $i‘
echo “$num * $i = $result”
done
sleep 5
clear
;;
2) exit;;
*) echo “***ERROR, NO HA ELEGIDO UNA OPCION VALIDA***”
sleep 3
clear
;;
esac
done
```

El siguiente ejemplo es una construción sencilla de un menú, lo llamaremos **menu.sh**:

```
#!/bin/bash
muestraopcionesmenuprin() {
echo ‘1) Fecha y hora’
echo ‘2) Calendario del més actual’
echo ‘3) Calendario del año actual’
echo ‘4) Calculadora de precisión arbitraria’
echo ‘5) Lista de usuarios conectados’
echo ‘6) Memoria libre del sistema’
echo ‘7) Carga del sistema’
echo ‘8) Ocupacion de todo el sistema de ficheros’
echo ‘9) Mi usuario es’
echo ‘0) Terminar’
echo
echo -e “Introduzca la opción deseada : ”
}
pausa () {
echo
echo -n “Pulse para continuar”
read
}
while true
do
muestraopcionesmenuprin
read OPT
clear
case $OPT in
3|7) echo “Para salir deberá pulsar ‘q’; pausa ;;
4) echo “Para salir deberá introducir ‘quit’“ ; pausa ;;
esac
echo ; echo
case $OPT in
0) exit ;;
1) date ; pausa ;;
2) cal ; pausa ;;
3) cal ‘date + %Y‘ | less ;; 4) bc ;;
5) who -iTH ; pausa ;;
6) cat /proc/meminfo ; pausa ;; # Podría usarse el comando free
7) top -s ;;
8) df ; pausa ;;
9) whoami ; pausa ;;
*) echo -e “Opcion erronea.” ; pausa ;;
esac
done
echo
```

Capítulo6

Automatización de tareas del sistema

6.1. Crontab

Existen en GNU/Linux una utilidad que no muchos conocen y que resulta a veces impresindible: **crontab**.

Crontab permite programar lo que se llaman crones, esto es, tareas o **scripts** que se ejecutarán en un momento determinado del tiempo de manera automática. El sistema GNU/Linux (y cualquier UNIX en general) comprueba regularmente, guiándose por el evento del reloj del sistema, si existe alguna tarea programada para ejecutarse y, en caso afirmativo, la ejecuta sin necesidad de nigún usuario lo haga explícitamente. El **cron** es usado normalmente para tareas administrativas, como respaldos, pero pude ser usado para cualquier cosa. El **cron** de GNU/Linux es un demonio **crond**. Su archivo de configuración se encuentra en **/etc/crontab**.

Algunas de las aplicaciones que se nos pueden ocurrir para algo así podrían ser:

1.- Apagar un quipo a la hora que nosotros queramos: por ejemplo mientras dormimos podemos dejarlo bajando cosas de internet haciendo que se apage solo por la mañana.

2.- Crear backups: podriámos programar un cron para que, a cierta hora de ciertos días de la semana o del mes, realizase un backup de nuestros datos de manera automática, sin tener que recordar nosotros el hacerlo.

3.- Poner mensajes recordatorio: para que nos salga en pantalla una ventana recordándonos que hagamos algo que podría olvidarsenos.

4.- Y muchísimas otras...

La herramienta **crontab** nos permite programar una tarea que se ejecute especificando:

1. La **hora** (0-23 minutos)
2. Los **minutos** (0-59)
3. El **día del mes** (1-31)
4. El **día de la semana** (0-7) tanto 0 como 7 representan al domingo
5. El **el mes** (1-12)

Cada uno de los 5 primeros campos puede contener:

Un asterisco (*****) Campo válido para cualquier valor. En cualquiera de éstos parámetros nos permitirá especificar mediante un ***** que deseamos que ocurra **todos** los minutos, las horas, los días del mes, los meses y los días de la semana.

Coma (**,**) separador de valores múltiples (o lista). Por ejemplo, podremos 1,15 en la columna del día del mes para que la tarea se ejecute cada día 1 y 15 del mes.

Guión (**-**) se usa para indicar **rangos** de valores, de manera que 1-4 en la columna del día de la semana significará de lunes a jueves.

Barra (/) indica paso de valor, por ejemplo, (*/3 en la columna mes, indica cada 3 meses; 0-59/2 en la columna minutos, indica cada 2 minutos).

6.1.1. Los crontabs de los usuarios

En los sistemas GNU/Linux, cada usuario tiene su propio crontab, incluyendo el superusuario root. En el crontab de cada usuario sólo se permitirá ejecutar tareas para las que ese usuario tenga permisos, por ejemplo, un usario que no sea **root** no podrá apagar el sistema. Cada usuario que programe una tarea con **crontab** tiene un archivo **/var/spool/cron/crontabs/nombre_usuario** que será su archivo de configuración.

6.1.1.1. Configuración de una tarea cron

El fichero de configuración principal de cron **/etc/crontab**, contiene las siguientes líneas siguientes:

```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root HOME=/
```

```
# run-parts
17 * * * * root run-parts /etc/cron.hourly
25 6 * * * root run-parts /etc/cron.daily
47 6 * * 7 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

Como podrá observar en el archivo **/etc/crontab** usa el **script run-parts** para ejecutar los scripts en los direcctorios:
/etc/cron.hourly,/etc/cron.daily,/etc/cron.weekly y /etc/cron.monthly cada hora, diriamente, semanalmente o mensualmente, respectivamente. Los archivos en estos directorios deben ser **scripts** de shell.

Las primeras cuatro líneas son variables que se usan para configurar el entorno en el que se ejecutan las tareas **cron**. Las primeras cuatro líneas indican lo siguiente:

SHELL es el shell bajo el cual se ejecuta el **cron**. Si no se especifica, tomará por defecto el indicado en la línea **/etc/passwd** correspondiente al usuario que este ejecutando **cron**.

PATH contiene o indica la ruta a los directorios en los cuales cron buscará el comando a ejecutar. Este **PATH** es distinto al **PATH** global del sistema o del usuario.

MAILTO es a quien se le envía la salida del coamando (si es que este tiene alguna salida). **Cron** enviará un correo a quien se especifique en esta variable, es decir debe se un usuario válido del sistema o de algún otro sistema. Si no se especifica, entonces cron enviará el correo al usuario propietario del comando que se ejecuta.

HOME es el directorio raíz o principal del comando **cron**, si no se indica, entonces la raíz será la que se indique en el archivo **/etc/passwd** correspondiente al usuario que ejecuta el **cron**.

6.1.1.2. Manejo del comando crontab

Para usar la funcionalidad de crontab, el comando nos ofrece de manera básica las siguientes posibilidades:

crontab -l nos muestra nuestra tabla de tareas programadas.
crontab -e sirve para editar nuestra tabla de tareas programadas.
crontab -r sirve para borrar nuestra tabla de tareas programadas.

6.2. Algunos ejemplos de cron

6.2.1. El formato de las entradas de crontab

Cuando añadimos una entrada a nuestro **crontab**, lo haremos en una sola líne y con el siguiete formato:

[minutos] [hora] [día] [mes] [día_de_semana] [comando]

Pongamos algunos ejemplos:

Para apagar el equipo a las siete y media de mañana del 10 de noviembre (debes ser root para ejecutar esto):
30 7 10 11 * /sbin/shutdwon -h now

Para hacer un .tar.gz de un directorio (como backup, por ejemplo), todos los días laborales a las 2:00 de la tarde:

0 14 * * 1-5 tar -czf /home/backups/backup.tar.gz /usr/importarate/

Para que cada hora los sábados y domingos se guarde en un fichero de log las conecciones de red abiertas en el sistema:

0 * * * 6,7 date >> /var/log/net_con.log; netstat -t >> /var/log/net_con.log

Cada 5 horas de lunes a viernes, se asegura que los permisos sean correctos en mi home:

1 *5 * * * 1-5 chmod -R 644 /home/drivera/*

Cada 15 minutos, todos los dias, horas y meses mostrará los ususrios que estan logueados en el sistema y guardará la salida en un archivo llamado /var/log/quienes_la_hora, por cada reporte deberá crear un archivo llamdo quienes con la fecha y hora en que se creó:

*/15 * * * * who -iH>/var/log/quienes_‘date +%F‘

Como se puede ver es súmamente fácil de manejar y muy potente. Espero que todo esto le sirva para dar sus primeros pasos en el mundo de la programación, así como para la buena administración de su sistema operativo. [1]