MANUAL DE SHELL SCRIPT

SISTEMAS OPERATIVOS EN RED

Mario García Alcázar

Índice de contenido

1 Introducción.

El lenguaje shell es un lenguaje de programación de alto nivel que por medio del uso de comandos Linux y sentencias de control de flujo, permite generar programas muy potentes destinados a automatizar tareas de administración de un servidor. Por ejemplo, podríamos crear programas que asegurasen que los usuarios no pudieran ejecutar diversos programas, que controlasen la seguridad y el buen rendimiento del sistema o que realicen automáticamente tareas pesadas y costosas. De hecho, es muy común que los administradores de servidores Linux, tengan una serie de programas Shell Script que se encargan de este tipo de tareas.

Existen diversas versiones de shell, no obstante, en este capítulo se tratarán las directrices básicas para construir programas sobre la shell de tipo bash, la cual, por otra parte es la versión más extendida.

2. Ejecución de un programa Shell Script.

Para ejecutar un guión o programa shell script, tenemos diferentes alternativas. Estas son las siguientes:

- Ejecutar el programa en la shell actual. Para ello, podemos:
  - Dar permisos de ejecución al fichero del programa y ejecutarlo en una consola. Esta es la forma más común.
  - Usar la orden . (punto). *Ejemplo:* . prueba
- Ejecutar el programa en un subshell. Para ello, a su vez, tenemos diferentes opciones:
  - Usar el comando **sh.** Este comando recibirá como parámetro, el nombre del fichero en el que se encuentra el programa a ejecutar. *Ejemplo:* sh prueba
  - Especificar una serie de ordenes entre paréntesis. *Ejemplo:* (date; pwd; ls)

Hay que tener en cuenta que si modificamos o damos valor a una variable de entorno en un subshell, NO se habrá definido o modificado en el shell principal.

Ejemplo:

```
a=1
(a=2)
echo $a # esto da como resultado 1
(a=2;echo $a) # esto da como resultado 2
```

3. Programando en Shell Script.

En este apartado trataremos las diversas características y recursos que nos ofrece el lenguaje shell.

3.1 Comentarios en guiones shell.

Se pueden insertar comentarios en los programas shell, utilizando el carácter #. Se ignora todo lo que existe desde el # hasta el final de la línea.

Ejemplos:

```
# ls -l esta línea no ejecuta nada.
rmdir prueba1 # esto borra el directorio prueba1
```

Nota importante:

En muchos scripts es necesario indicar el tipo de shell con le que debe ejecutarse dicho programa. Esto es imprescindible si el script va a lanzarse automáticamente al inicio del sistema.

Para indicar este dato, se pone al principio del shell la siguiente línea comentada:

#!/bin/bash

3.2 Argumentos en los programas shell.

Podemos pasar argumentos por la línea de comandos a un programa shell, de forma similar a como se hace en otros lenguajes como, por ejemplo C. Simplemente basta con escribir dichos argumentos, detrás del nombre del programa a ejecutar.

Ejemplo:

```
sumar 1 2 4 33 # el programa se llama sumar y se le pasa como argumentos 1, 2, 4 y 33.
```

3.3 Variables en los programas.

En Shell Script, como en cualquier lenguaje de programación, se permite el uso de variables. En concreto, podemos identificar dos tipos de variables:

1. **Variables comunes definidas por el programador.**

   Permiten almacenar y operar con valores.

   Toman valor por medio del operador de asignación =. *Ejemplo*: A=10

   Para consultar su valor debemos anteponer el carácter $. *Ejemplo*: echo $A

2. **Variables especiales.**

   Permiten manejar diferentes elementos, como por ejemplo los argumentos de los que se hablo en el punto anterior. Estas variables son las siguientes:

   - **$#** indica el numero de argumentos que se pasa al guión. En el ejemplo anterior seria 4.
   - **$1, $2, $3** hacen referencia al primer, segundo y tercer argumento del guión. En el ejemplo anterior tendrían como valor 1, 2 y 4.
   - **$0** es es nombre del programa shell que estamos ejecutando.
   - **$*** hace referencia a todos los argumentos de la línea de órdenes. En el ejemplo que estamos tratando seria 1 2 4 33
   - **$?** contiene el resultado de la última orden ejecutada.
   - **$!** contiene el PID del último proceso hijo creado.
   - **$$** contiene el PID del proceso generado al ejecutar el programa shell.

3.4 El comando shift.

Este comando, permite desplazar los parámetros de entrada de un guión de derecha a izquierda, de forma que $1 cambia su valor original por el de $2, $2 pasa a tener el valor de $3 y así sucesivamente.

Este comando puede ser interesante para acceder a los parámetros de entrada de scripts, de una forma cómoda.

Ejemplo de uso:

```
echo $1 # Escribe por la salida estándar, el valor del primer parámetro.
shift
echo $1 # Escribe por la salida estándar, el valor del segundo parámetro.
```

3.5 Expansión de variables.

A la hora de operar con variables en shell script, hay que tener en cuenta que shell sustituye estas por el valor que tuvieran asignado. No obstante, podemos realizar ciertas operaciones especiales sobre ellas, por medio del operador {}.

Estas operaciones son las siguientes:

1. ***${nombre_variable:-valor_defecto}***

   Si la variable *nombre_variable* no ha sido definida en el script, se usará el *valor_defecto*. Si queremos evaluar el valor de una variable de entorno, previamente se deberá haber ejecutado el comando export sobre dicha variable, para que esté accesible al script y a todos los shell del usuario.

   Ejemplos:

```
echo ${b:-hola} # Como b no está definida en el script, muestra la cadena hola.
echo $b # Muestra una cadena vacía.
```

2. ***${nombre_variable:=valor_defecto}***

   Si la variable *nombre_variable* no ha sido definida en el script, se le asignará como valor, el *valor_defecto*. Si queremos evaluar el valor de una variable de entorno, previamente se deberá haber ejecutado el comando export sobre dicha variable, para que esté accesible al script y a todos los shell del usuario.

   Ejemplos:

```
echo ${b:=hola} # Como b no está definida en el script, se le asigna valor 'hola'
                # y se muestra por la salida estándar.
echo $b # Muestra la cadena hola
```

3. ***${nombre_variable:?mensaje}***

   Si la variable *nombre_variable* no ha sido definida en el script, se le muestra por la salida estándar el valor del campo *mensaje* y finaliza la ejecución del script. Si queremos evaluar el valor de una variable de entorno, previamente se deberá haber ejecutado el comando export sobre dicha variable, para que esté accesible al script y a todos los shell del usuario.

   Ejemplo:

```
echo ${c:-la variable no tiene valor} # Como c no está definida en el script, se muestra la
                                      #  cadena “la variable no tiene valor” y acaba
                                      #  la ejecución del guión.
```

3.6 Comillas.

El lenguaje Shell, permite el uso de tres tipos de entrecomillado para llevar a cabo el tratamiento de cadenas y expresiones.

Los tipos de entrecomillado son los siguientes:

- **Comilla invertida ` `**

  Se sustituye el valor de las variables internas y ejecuta el contenido.

  Ejemplos:

```
res= `ls -l` # Se guarda en la variable res el resultado de ejecutar comando ls -l
a=/etc
res1= `ls -l $a` # Se guarda en la variable res1 el resultado de ejecutar el comando ls -l /etc
```

- **Sin comillas o con comilla doble “ “**

  Se sustituye el valor de las variables internas, pero no se ejecuta nada.

  Ejemplo:

```
a=/etc
res=“ls -l $a”        # Guarda la cadena “ls -l /etc” en la variable res.
```

- **Con comilla normal ' '.**

  Se toma como un texto fijo. No se interpretarían las variables de entorno.

  Ejemplo:

```
a=1
res= '$a'          # Guarda la cadena “$a” en la variable res.
```

3.7 El comando read.

Permite leer datos introducidos por el usuario por pantalla.

Su formato es sencillo: read nombre_variable

Ejemplo: read a1

3.8 El comando exit.

Finaliza la ejecución del programa Shel Script y devuelve como resultado de este, el *número_resultado* que se especifica como parámetro.

Su formato es: exit *numero_resultado*.

Ejemplo: exit 1

Notas: Normalmente un valor devuelto igual a cero indica que el programa terminó correctamente y un valor devuelto no cero,indica que ocurrió algún error.

Además, si un guión lanza la ejecución de otro, puede obtener su valor devuelto por medio de la variable $? que se comentó anteriormente.

3.9 Operaciones aritméticas.

Shell script permite realizar operaciones aritméticas mediante el uso del comando **expr**.

La orden *expr* toma los argumentos dados, los evalúa e imprime el resultado sobre la salida estándar. Cada término de la expresión debe ir separado por espacios en blanco.

La orden *expr* puede utilizar los siguientes operadores: +, -, *, /, % (módulo). No obstante, al ser el carácter * el símbolo comodín del shell, para indicar que se quiere multiplicar deberá ir precedido por el carácter \.

Hay que indicar también que *expr* sólo opera sobre enteros y no sobre números decimales.

Ejemplos:

```
i= `expr $1 + 1` # Se suma 1 al primer parámetro.
k=`expr $1 \* 5`  # Se multiplica el primer parámetro por 5
```

Otra forma mucho más rápida de realizar operaciones aritméticas es por medio de la siguiente sintaxis:

$[operación] # Sin comillas de ejecución

Ejemplos:

```
i=$[1+2]    # i=3
i=$[$i+1]   # i=i+1
```

```
echo $[$i/2]  # mostraria $i entre 2
j=$[$i*5]   # j=$i*5
```

3.10 Ejecución condicional.

Como en muchos otros lenguajes, en shell script, se permite la ejecución condicional de partes del programa. Para construir esta ejecución condicional, se usa los comando if y case.

3.10.1 El comando if.

Su formato es el siguiente:

if *órden_comparación*
then órdenes
[else órdenes**]**
fi

El funcionamiento de este comando es el siguiente. Si el valor de *órden_comparación* es igual a cero, se ejecutan las ordenes de la parte de then. En caso contrario se ejecutan las ordenes de la parte de else, en caso que existan.

La ***orden_comparación*** puede expresarse de diferentes formas:

1. Mediante la introducción de un comando común.

   Este comando se ejecutará y se tomará como orden_comparación, el su valor devuelto. Si el comando funciona correctamente, su valor devuelto será cero, y en caso contrario su valor devuelto será distinto de cero. Puede verse un ejemplo en el *ejemplo 2* de este apartado.

2. Mediante el uso de la orden **test** o su equivalente **[]**.

   test evalúa una expresión; si la expresión es verdad, devuelve cero; si la expresión no es verdad, devuelve un estado no cero.

   Su formato sería uno de los siguientes:

   - **test** *comparación*

     Ejemplo: test $1 = 'hola'

   - **[** *comparación* **]**

     Ejemplo: [$1 = 'hola']

Por medio de esta orden, **podemos realizar diferentes comprobaciones**, en función de lo que se indique en el campo *comparación*. Seguidamente se especifican las más comunes:

Comprobaciones de enteros.

n1 -eq n2	Verdadero si los enteros n1 y n2 son iguales
n1 -ne n2	Verdadero si los enteros n1 y n2 no son iguales
n1 -gt n2	Verdadero si el entero n1 es mayor que el n2
n1 -ge n2	Verdadero si el entero n1 es mayor o igual que el n2
n1 -lt n2	Verdadero si el entero n1 es menor que el n2
n1 -le n2	Verdadero si el entero n1 es menor o igual que el n2

Comprobaciones de cadenas de texto.

-z cadena	Verdadero si la longitud de la cadena es cero.
cadena1 = cadena2	Verdadero si las dos cadenas son iguales
cadena1!= cadena2	Verdadero si las dos cadenas no son iguales
cadena	Verdadero si la cadena no es nula

Comprobaciones de ficheros.

-a archivo	Verdadero si existe el archivo
-r archivo	Verdadero si existe el archivo y puede leerse
-w archivo	Verdadero si existe el archivo y puede escribirse
-x archivo	Verdadero si existe el archivo y es ejecutable
-d archivo	Verdadero si es un directorio
-h archivo	Verdadero si es un enlace simbólico
-c archivo	Verdadero si existe el archivo y es un archivo especial de caracteres
-b archivo	Verdadero si existe el archivo y es un archivo especial de bloque
-s archivo	Verdadero si existe el archivo y su tamaño es mayor de cero

Entre las condiciones y los operandos siempre tiene que haber un espacio en blanco.

Hay que indicar también que en mismo if podemos combinar distintas ordenes de comparación, mediante los operadores **&& (AND), || (OR) y ! (NOT).** Su sintaxis es la siguiente:

```
[ condición1 ] && [ condición2]
[ condición1 ] || [ condición2]
! [ condición ]
```

Pueden verse un ejemplos de esto en los *ejemplos 4* y 5 de este apartado.

Ejemplos:

Ejemplo 1

```
if [ $1 = 'a' ]
then
echo $1
ls -l >res
fi
```

Ejemplo 2

```
if  `mkdir PRUEBA`
then
echo “el directorio se creó correctamente”
else
echo “el directorio no pudo crearse”
exit 1
fi
```

Ejemplo 3

```
if [ $# -lt 3 ]
then echo “el número de parámetros es menor de tres”
exit 1
else echo “el número de parámetros es correcto”
fi
```

Ejemplo 4

```
if [ $1 -gt 1 ] && [ $1 -lt 5 ]
then
   echo 'El paŕametro $1 es mayor que 1 y menor que 5'
fi
```

Ejemplo 5

```
if ! [ $1 = “a” ]
then
  echo 'El parámetro $1 es distinto de la letra a'
fi
```

3.10.2 El comando case.

Su formato es el siguiente:

```
case cadena
in
lista_patrones)
   orden 1
   orden 2
   orden n
   ;;
[lista_patrones)
   orden 1
   orden 2
   orden n
   ;;]
esac
```

En su funcionamiento, permite ejecutar una secuencia de comandos, si el valor de *cadena*, está en alguna de las *lista_patrones* que se especifican en el comando.

Ejemplo:

```
case $mes in
10 | 11 | 12)
echo “Estamos en octubre, noviembre o diciembre”
exit 0
;;
[1-8] | 9)
echo “Estamos en uno de los nueve primeros meses del año”
exit 0
;;
*) # Esto identifica cualquier valor de cadena, por lo que sirve como valor por defecto.
exit 1
;;
esac
```

3.11 Bucles.

Un bucle es una estructura que permite repetir varias veces un mismo segmento de código, normalmente mientras se cumpla una determinada condición.

Shell script, permite la utilización de bucles por medio de tres sentencias básicas.

3.11.1 El bucle for.

Este bucle repite la ejecución de las ordenes indicadas, para cada uno de los valores que se proporcionan en la *lista de valore*s. De hecho, en cada iteración se asigna uno de esos valores a la *variable* y se ejecuta el conjunto de órdenes.

Su formato es el siguiente:

```
for variable in lista_valores
do
    órdenes
done
```

Ejemplos:

Ejemplo 1

```
# Este programa hace una iteración por cada valor de la lista 1 2 3 4 5
for k in 1 2 3 4 5
do
   # ordenes del bucle
done
```

Ejemplo 2

```
# Este bucle realiza una iteración por cada fichero existente en el directorio /etc
for i in `ls /etc`
do
  echo “fichero $i”
  # operaciones sobre el fichero ....
done
```

3.11.2 El bucle while.

Este comando permite repetir una serie de órdenes mientras se cumpla una condición. Las condiciones se definen con el mismo formato que se explicó para el comando **if**. Su formato es el siguiente:

```
while condición
do
    órdenes
done
```

Ejemplo:

```
# Este programa muestra por pantalla los números de 1 al 5
i=1
while [ $i -lt 5 ]
do
    echo $i
    i=$[$i+1]
done
```

3.11.3 El bucle until.

Permite repetir una serie de órdenes hasta que se cumpla la condición indicada. Como puede verse su funcionalidad es muy parecida al bucle while, de hecho, normalmente puede usarse uno u otro indistintamente. Las condiciones se definen con el mismo formato que se explicó para el comando **if .** Su formato es el siguiente:

```
until condición
do
    órdenes
done
```

Por último, hay que hablar de dos ordenes que pueden ser interesantes dentro de un bucle. Estas son las ordenes *break* y *continue*.

- *break* sale del bucle que lo engloba.
- *continue* controla la vuelta al principio del bucle más pequeño que lo engloba.

7.12 Funciones.

Como la mayoría de los lenguajes de programación de alto nivel, shell script, permite la creación de funciones dentro de un guión, para hacer más fácil y legible el código.

Para crear una función se utiliza la siguiente sintaxis:

```
function nombre_función
{
  lista de ordenes
}
```

Para invocarla, basta con escribir su nombre, seguida de los parámetros adecuados:

nombre_función p1 p2 p3 ....

Por último, hay que indicar que todas las variables que se definan dentro de una función, son globales a todo el guión. Si se desea crear una variable local a una función, basta con utilizar el comando **typeset,** definiendo la variable con la siguiente sintaxis **typeset nombre_variable**

Ejemplos:

Ejemplo 1

```
function hola
{
    echo Hola $1
}

# Programa principal
hola Mario  # Esto muestra “Hola Mario” por la salida estándar
hola Juan    # Esto muestra “Hola Juan” por la salida estándar
```

Ejemplo 2

```
function aux
{
   typeset k
   k=2
}

# Programa principal
```

```
aux
echo $k # Esto imprimirá una cadena vacía, ya que la variable k es local a la función
```

7.13 Recursividad.

El lenguaje shell permite instrumentar algoritmos recursivos. Para ello, podemos implementar dicha recursividad por medio de funciones (como en cualquier otro lenguaje de programación) o usando la orden **sh** seguida del nombre del guión que estamos ejecutando.

Sh ejecuta un comando o programa shell script que se le pase por parámetro, por tanto puede usarse para implementar soluciones recursivas, aunque su manejo para este fin es mucho mas complejo que el de las funciones. Por esto, en este libro recomendamos el uso de funciones en caso que sea necesaria una solución recursiva.

Seguidamente se muestran dos ejemplos de recursividad:

Ejemplo 1

```
# Cálculo recursivo del factorial de un número con el comando sh
if [ $# -lt 1 ]
then
 echo "Número de parámetros incorrecto"
 exit 1
fi

if [ $1 -gt 1 ]
then
  if [ $# -eq 1 ]
  then
    acumulado=$1   # Si es la primera llamada, el valor acumulado
                   # es el número a factorizar
  else
    acumulado=`expr $1 \* $2` # En caso contrario el valor acumulado
                              # es el acumulado anterior, por el factor
                              # actual
  fi
```

```
  numero=`expr $1 - 1`
  sh $0 $numero $acumulado  # llamada recursiva
else
  echo "El resultado final es $2"
  exit 0
fi
```

Ejemplo 2

```
# Factorial con función recursiva.
function factorial
{
   i=$1
   if [ $i -gt 2 ]
   then
      res=`expr $res \* $i`
      i=`expr $i - 1`
   fi
}

# Programa principal
res=1
factorial $1
echo El resultado es $res
```

3.14 Arrays.

Bash shell ofrece la posibilidad de usar arrays de valores utilizando una sintaxis con corchetes para los subíndices. La única limitación en este caso es que el índice de los mismos debe estar entre 0 y 511, por lo que tan sólo son posibles vectores de hasta 512 componentes.

Para crear un vector basta con asignar un valor a uno de sus componentes. Los restantes se irán creando de forma dinámica conforme se vayan necesitando. Para acceder a un componente particular se usa la sintaxis ${nombre_vector[i]}.

Además para ver el número de elementos de un vector, se escribe ${#nombre_vector[*]}

Ejemplos:

```
amigos[0]=Juan
amigos[1]=Luís
amigos[2]=María
amigos[3]=Ana
${amigos[0]} está casado con ${amigos[2]} # Juán está casado con María
echo Tengo ${#amigos[*]} en la agenda        # Tengo 4 amigos en la agenda
echo Mis amigos son ${amigos[*]}             # Mis amigos son Juán Luís María Ana
```

3.15 Cuadros de diálogo.

Aunque shell script es un lenguaje orientado al trabajo en entornos de texto, es posible "adornar" la presentación de nuestros programas con cuadros de diálogo para mostrar mensajes e interactuar con el usuario.

Para ello, usaremos el comando **dialog**, el cual usa tiene muchas opciones para configurar el aspecto y los botones a mostrar. Algunas de estas funciones son:

--backtitle → Título de la ventana

--title → Título de la caja de texto interna a la ventana.

--msgbox → Contenido y dimesiones (ancho y alto) de la caja de texto interna a la ventana, la cual mostrará además un botón de "ok"

--yesno → Como la opcion msgbox, pero en vez de mostrar el botón "ok" muestra los botones "yes" y "no". Los resultados de estos botones sse pueden obtener a partir de la variable $? donde el valor 0 será yes y el 1 será no.

--inputbox → Como la opción msgbox, pero muestra un cuadro de texto para que el usuario introduzca un valor. El datos introducido por el usuario saldrá por la salida de error y en la variable $? tendremos un 0 si ha pulsado al botón "ok" o un 1 si ha pulsado "cancelar".

--passwordbox → Como la opción anterior pero no se muestra el texto mientras se escribe.

--menu → Permite mostrar un menú. Hay que indicar el título del menú, usu dimensiones (alto de la ventana, ancho de la ventan y dimensión del menú) y un identificador de cada opción seguido por el texto de esta que se va mostrar. **(Ver ejemplo)**

Ejemplos:

```
# Ventana de confirmación con botón “OK”
dialog --backtitle "VENTANA DE CONFIRMACION" --title "SALUDO" --msgbox "HOLA" 9 20

# Ventana de yes/no
dialog --backtitle "VENTANA DE CONFIRMACION" --title "SALUDO" --yesno "Aceptar?" 9 20
echo $?

# Cuadro de texto para la introducción de datos
dialog --backtitle "Programa 1" --inputbox "Introduzca el nombre de un usuario" 9 20 2> aux

res=$?
usu=`cat aux`
if [ $res -eq 0 ]
then
  echo echo el texto introducido es $usu y el botón pulsado es ACEPTAR
else
  echo el texto introducido es $usu y el botón pulsado es CANCELAR
fi



# Menú gráfico para un programa
dialog --menu "MENU DEL PROGRAMA" 20 30 20 1 "Opcion 1" 2 "Opcion 2" 2> aux

res=$?
op=`cat aux`

if [ $res -eq 0 ]
then
  echo La opcion elegida $op y el botón pulsado es ACEPTAR
else
  echo La opcion elegida $op y el botón pulsado es CANCELAR
fi
```