Documentos Aula SUN UCM

CREACIÓN DE SCRIPTS EN LINUX

26 Mayo 2008

Autores:

Sergio Velasco

Alicia Martín

Aula SUN UCM de Software Libre

INDICE

CREACIÓN DE SCRIPTS EN LINUX

Definición de Script: Un **script** es un archivo que incluye un conjunto de comandos. Son ejecutados desde la primera línea hasta la última (de forma secuencial).

1- ¿QUÉ ES EL SHELL SCRIPT?

Un Shell Script es un script para la shell de comandos (terminal). Para crear un script vasta con un editar un fichero nuevo y en el nombre poner *.sh* Ejemplo: *HolaMundo.sh*

Un vez creado, empezamos a editarlo. Se puede utilizar un editor de textos gráfico como ***gedit*** o un editor en terminal como vim, nano o emacs.

En la primera línea del script se debe indicar que shell que vas a usar (*/bin/bash/* , */usr/bin/perl* , etc) Aunque da igual la que uses lo importante es el contenido:

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

#! Se conoce con el nombre de ***Sha Bang***.

Se denomina "*sha-bang*" a la secuencia **#!** con la que se inician los scripts. Su función es indicarle al sistema que se trata de un conjunto de comandos para que sean interpretados. En realidad, es un *número mágico* de dos bytes. El *número mágico* es un marcador especial para indicar el tipo de archivo, en este caso, indica que se trata de un script de shell ejecutable.

Para introducir comentarios se debe poner #. Por cada linea que deseéis poner un comentario, lo primero que debe tener es #. Es importante añadir comentarios comentando la utilidad del script o las variables que se crean.

2- ¿CÓMO EJECUTAR UN SCRIPT?

Antes de poder ejecutarlo, debemos darle permisos de ejecución. (+x) por ello, haremos uso del comando **chmod** y damos permisos de ejecución, si se desea, se pueden dar a todos los usuarios y grupos.

`chmod 755 /ruta_del_script` Para el usuario propietario

`chmod 777 /ruta_del_script` Para cualquier usuario

Una vez hecho todo lo anterior, usaremos:

./nombredelscript.sh

Pero también podemos usar si es un shell script:

sh nombredelscript.sh

3- CÓDIGO DE UN SCRIPT

Ya tenemos el script creado, le hemos puesto la cabecera y le hemos cambiado los permisos, ya solo falta meter el código.

Vamos a empezar desarrollando lo esencial para ir desarrollando estructuras más complejas:

Lo primero es saber cómo dar valor a una variable. Es tan sencillo como poner:

```
nombre_variable=valor
```

Si deseas guardar la salida de un programa solo tienes que ponerlo entre tildes invertidas:

```
nombre_variable=`comando`
```

Tambien hay un comando que lee por teclado las variables (**read**). Para ponerlo es:

```
read [opciones] nombre_variable1 nombre_variable2 nombre_variableN

ejemplo:
read -p "Introduce el nombre y los apellidos" nombre apellidos
```

Tiene un montón de opciones pero estas son las más importantes y usadas:

-n num_car : Número máximo de caracteres que puedes introducir por teclado
-p "frase" : Te muestra por pantalla una frase para tu saber que debes introducir
-d "delimitador" : Especificas cual va a ser el delimitador, es decir si dices que el delimitador sera ";" pues todo lo que venga antes de un ";" lo cogerá una variable y todo lo que venga después de ese delimitador hasta el próximo ";" lo cogerá otra variable.

Cuando queremos utilizar el valor de una variable en el código, nos referiremos a éste como:

```
$nombre_variable
```

EJEMPLOS 1

Estos son algunos de los ejemplos de esta primera parte. Para ejecutarlos sólo hay que crear un archivo .sh y copiar las letras negras, lo azul es la sugerencia para el nombre del script.

A lo largo de los ejemplos se introducen algunos comandos básicos de Linux.

HolaMundo.sh

```
#! /bin/bash
clear
echo "Hola mundo ,este es mi primer script"
```

ScriptUno.sh

```
#! /bin/bash
clear
nombre="Perico"
apellidos="Palotes"
echo "Te llamas $nombre $apellidos"
```

Fecha.sh

```
#! /bin/bash
clear
fecha=`date | cut -d " " -f 1,2,3`
hora=`date | cut -d " " -f 4`
echo "Hoy es $fecha y son las $hora"
```

OtroScript.sh

```
#! /bin/bash
clear
# IFS es una variable de entorno que determina el delimitador de
#campos
# (que por defecto vale " "),en este script queremos
# cambiarlo a "," para escribir por teclado nombre,apellidos
IFS=","
read -p "Introduce el nombre,apellidos : " nombre apellidos
echo "El nombre es $nombre y los apellidos son $apellidos"
```

3.1- Depuración

Depuración de programas es el proceso de identificar y corregir errores de programación.

En inglés se le conoce como *debugging*, ya que se asemeja a la eliminación de *bichos* (bugs), manera en que se conoce informalmente a los errores de programación. Se dice que el término *bug* proviene de la época de las computadoras de bulbos, en las cuales los problemas se generaban por los insectos que eran atraídos por las luces y estropeaban el equipo.

Depurar el código sirve para ver como se ejecuta paso por paso el script, que valores toman sus variables, si has cometido un fallo saber en que parte del código ha sido, etc. No es algo que se deba hacer obligatoriamente por cada script realizado (sería demasiado pesado) pero te facilita mucho a la hora de buscar el problema que te surja. Hay dos formas :

1.O bien en la línea #! /bin/bash -x | -v
-x –> Te muestra las instrucciones antes de ejecutarlas por consola y sustituyendo las variables
-v –> Te muestra todo pero sin sustituir las variables
2.O Mediante **set**
Cambiando las opciones de ejecución de la shell a lo largo del script, también con las opciones -x | -v

3.2- Estructuras condicionales

La estructura básica de una condición sería:

```
if condición
    then
        comando1
        ...
        ...
else
        comando1
        ...
        ...
fi
```

Como veis la condición si (*if*) se cierra con su correspondiente *fi* que al parecer un juego de palabras es fácil de recordar.
Entonces si la condición se cumple entraría por el *then*, en caso de que no, por el *else*.
Pero este es un método simple, luego tenemos uno más complejo con *if anidados*, sería:

```
if condición
    then
        comando1
elif condición
    then
comando
elif condición
then
        ...
fi
```

Ahora lo que hace es evaluar la condición, si es verdadera entra por el *then*, pero si no y se da el caso de otra condición entraría por el *elif*, los elif no se cierran, solamente el *fi* final corresponde a la apertura del *if*.

La condición es cualquier cosa que de un return (que devuelva) algo que sea 0 o verdadero.

Notese el uso del archivo /dev/null como archivo vacío para comprobar algunas condicionales.

Vemos como podemos hacer un script para comprobar que realmente la ruta es un directorio.

CompruebaDirectorio.sh

```
#! /bin/bash
clear
if `cd /tmp/prueba/ >/dev/null`
then
echo "Pues si, es un directorio y contiene…"
ls -l
else
echo "Pues va a ser que no es un directorio"
fi
```

Este script es un claro ejemplo de como comprobar si un usuario y un grupo existen en el sistema, vemos también el uso que se le da al *elif*.

ExisteGrupoUsuario.sh

```
#! /bin/bash
clear
read -p "Introduce usuario… " user
read -p "Introduce grupo… " group
if `grep -e "^$user:.*" /etc/passwd >/dev/null`
    then
    if `grep -e "^$group:.*" /etc/group >/dev/null`
        then
        echo "Usuario y grupo ya existen en el sistema"
    fi
elif `grep -e "^$group:.*" /etc/group >/dev/null`
    then
    echo "usuario no existe, grupo si!!"
else
    echo "Ni grupo ni usuario existen"
fi
```

Condicionales case

Se estructuran de la siguiente forma:

```
case expresion in
caso1) comandos ;;
caso2) comandos ;;
*) comandos ;;
esac
```

Para aquellos que sepan de programación, funciona igual que los ***switch***, esta estructura sirve para dependiendo de la expresión se cumple un determinado caso, es decir, según sea el valor de esa expresión se hará un caso u otro. Esto se ve mejor con un ejemplo:

tecla.sh

```
#! /bin/bash
clear
read -n 1 -p "Pulsa una tecla " tecla
case $tecla in
[a-z,A-Z]) echo "Ha introducido una letra" ;;
[0-9]) echo "Ha introducido un numero" ;;
*) echo "Ha introducido un caracter especial" ;;
esac
```

Este script sirve para comprobar que tecla has pulsado, primero pide por teclado la tecla que será guardada en la variable tecla, y se usa en el ***case***. Si la tecla es una letra se hará todos los comandos que hayan en el caso `[ a-z,A-Z ]`(ya que una letra puede ser cualquiera, hace falta abarcar a todas las posibilidades por eso se pone el intervalo de a-z y A-Z), si la tecla es un número se hará todos los comandos que haya en el caso `[0-9 ]`(ya que puede ser cualquier número, hace falta abarcar a todas las posibilidades por eso se pone el intervalo de 0-9) y si la tecla no es un número ni una letra, se ejecutan todos los comandos del caso *****.

Se pone el ***** cuando no se va a cumplir el resto de casos. Es muy importante saber que las dobles comas (**;;**)se pone obligatoriamente al final de cada caso ya que marcan el final de ese caso, sino el programa no funcionará. Cuando hagáis una condicional debéis poner al final el **esac** ya que es el cierre del **case**. Se suele utilizar mucho esta estructura para mostrar un menú al usuario. Ejemplo:

ejmploMenu.sh

```
#! /bin/bash
clear
echo "1.Ejemplo de menu uno"
echo "2.Ejemplo de menu dos"
read -n 1 -p "Introduce una opcion" opcion
case $opcion in
1) exit 1 ;;
2) exit 2 ;;
*) echo "No has introducido ni un dos ni un uno" ;;
esac
```

Tener que poner tantos ***echo*** es bastante molesto, por eso hay un comando que te ahorra hacer ese esfuerzo (***select***):

```
select variable in "caso 1" "caso 2" "caso N"
do
break
done
case $variable in
"caso 1") comandos ;;
"caso 2") comandos ;;
"caso N") comandos;;
esac
```

El *break* sirve para que solo te muestre una vez el menú. Cuando usas ***select*** no hace falta pedir que introduzcas nada, ya que eso lo hace automáticamente. El prompt que te muestra ***select*** es **#?** pero se puede cambiar, poniendo otro valor a PS3. Por último veamos el ejemplo anterior con ***select***:

ejemploMenu.sh

```
#! /bin/bash
clear
PS3="Introduce una opcion"
select opcion in "1.Ejemplo de menu uno" "2.Ejemplo de menu dos"
do
break
done
case $opcion in
1) exit 1 ;;
2) exit 2 ;;
*) echo "No has introducido ni un dos ni un uno" ;;
esac
```

3.3- Operaciones algebraicas

En este post veremos el uso de **expr** el cual utilizaremos en multitud de ocasiones para hacer operaciones aritméticas, u operaciones con cadenas y un sinfín de cosas…

```
expr operación_a_evaluar
```

1.**Operaciones numéricas**:

expr num1 + num2 –> Devuelve la suma de num1 + num2
expr num1 - num2 –> Devuelve la resta de num1 - num2
expr num1 * num2 –> Devuelve el producto de num1 * num2
expr num1 / num2 –> Devuelve la división de num1 / num2
expr num1 >= num2 –> Devuelve 0(true) si num1 >= num2
expr num1 > num2 –> Devuelve 0(true) si num1 > num2
expr num1 <= num2 –> Devuelve 0(true) si num1 < num2

expr num1 < num2 –> Devuelve 0(true) si num1 < num2
expr num1 != num2 –> Devuelve 0(true) si num1 es distinto de num2

ExprEjemplo.sh

```
#! /bin/bash
clear
PS3="Introduce Opción: "
select opcion "suma" "resta"
do
break
done
read -p "Introduce dos números… " num1 num2
case $opcion in
suma)
echo "La suma de $num1 y $num2 es … `expr $num1 + $num2` " ;;
resta)
echo "La resta de $num1 y $num2 es … `expr $num1 - $num2` " ;;
esac
```

2.**Operaciones con cadenas**:

expr length cadena –> Nº de caracteres de esa cadena
expr index cadena_donde_busca cadena_a_buscar –> Devuelve la posición donde encuentra los caracteres a buscar dentro de la cadena, si no, devuelve un 0

ExprEjemplo2.sh

```
#! /bin/bash
clear
frase="Buenos días, estamos aprendiendo a programar"
echo "La longitud de la cadena es… `expr length $frase`"
read -p "Introduce alguna cadena que buscar " buscar
if [ ! -z $buscar ]
then
echo "Los caracteres $buscar se encuentran en la pos `expr index
$frase $buscar`"
fi
```

3.**expr match cadena_donde_busca patrón_caracteres_buscar** –> Funciona igual que el index pero los caracteres pueden ser una expresión regular

ExprEjemplo3.sh

```
#! /bin/bash
clear
frase="Buenos días, estamos aprendiendo a programar"
echo "La longitud de la cadena es… `expr length $frase`"
read -p "Introduce alguna cadena que buscar " buscar
if [ ! -z $buscar ]
then
echo "Los caracteres $buscar se encuentran en la pos `expr match
$frase [a-z,A-Z]*$buscar` de la frase"
fi
```

expr substr cadena posición n_caracteres –> Extrae de una cadena una subcadena de la cadena indicada a partir de esa posición y de longitud ese número de caracteres

ExprEjemplo4.sh

```
#! /bin/bash
clear
read -n 9 -p "Introduce tu NIF (FORMATO: 00000000A)… " nif
echo "El DNI de $nif es `expr substr $nif 1 8`"
echo "La letra del $nif es `expr substr $nif 9 1`"
```

4- BUCLES FOR

```
for variable in valor1 valor2 … valorN
do
comando1
…
[ break | continue ]
done
```

El bucle se ejecuta para cada uno de los valores que toma la variable en esa lista.

Break: Rompe el bucle y no da más opción a que la variable se ejecute.
Continue: Salta al siguiente valor de la lista.

Planetas.sh

```
#! /bin/bash
clear
for planeta in "Jupiter 10" "Venus 30" "Saturno 15" "Mercurio 1" Luna
Tierra
do
if [ "$planeta" = Tierra ]
then
break
elif [ "$planeta" = Luna ]
then
continue
else
echo "El planeta $planeta 0.000.000 Km del Sol"
fi
done
echo "fin del script"
```

Y su ejecución sería:

```
El planeta Júpiter 10 0.000.000 Km del Sol
El planeta Venus 30 0.000.000 Km del Sol
El planeta Saturno 15 0.000.000 Km del Sol
El planeta Mercurio 1 0.000.000 Km del Sol
fin del script
```

En *continue* puedes especificar el bucle en el que quieres que la variable salte, hablo de bucles anidados. Por defecto, poner *continue* es como poner *continue 1*, la variable del actual bucle salta al siguiente valor, veamos un ejemplo de ello:

PlanetasBacteria.sh

```
#! /bin/bash
clear
for planeta in Venus Saturno Jupiter Tierra Luna Sol Marte
do
for bacterias in Pepito Anemo Coli Streptos
do
if [ "$planeta" = "Tierra" ]
then
exit 45
elif [ "$bacterias" = "Pepito" ]
then
continue 2
else
echo "El planeta $planeta tiene la bacteria ….. $bacterias"
fi
done
done
```

El resultado de su ejecución sería:

```
El planeta Venus tiene la bacteria ….. Pepito
El planeta Saturno tiene la bacteria ….. Pepito
El planeta Júpiter tiene la bacteria ….. Pepito
```

Otra de las particularidades de este bucle es que puede ejecutarse a la forma de un bucle en java

```
(( valores ))

for(( variable=valor; condición; incremento ))
do
comando
…
[ break | continue ]
…
done
```

Es igual que en java pero con doble paréntesis.

Contador.sh

```
#! /bin/bash
clear
read -p "Introduce un numero" numero
for(( a=0; a <= $numero; a++ ))
do
echo -e "$a \n"
done
```

HacerPing.sh

```
#! /bin/bash
clear
read -p "¿Cual es el número de sunny(31,32,33,35,37)? " aula
if [ $aula -ne 31 -o $aula -ne 32 -o $aula -ne 35 -o $aula -ne 37 ]
then
echo "Valor incorrecto"
exit 1
else
for(( equipo=1; equipo <= 16; equipo++ ))
do
ping -c 1 192.168.$aula.$equipo | grep "0% lost" >/dev/null/
done
fi
```

5- BUCLES WHILE

Antes de aprender a hacer un script utilizando los bucles while, debemos conocer el funcionamiento del comando test, pues es muy usado en este tipo de bucles.

5.1- Comando test

Este comando sirve para expresar condiciones y evaluarlas, si son correctas origina códigos de salida = **0** y si son falsas = **1**

El comando pretende abreviar un poco en algunos casos, por eso se suele utilizar su forma corta:

```
test expresiones ––> [ expresión ]
```

Hay que tener en cuenta que la forma es:

```
[(espacio)expresión(espacio)]
```

Ya que si no se ponen los espacios en blanco daría lugar a error.

Un ejemplo de su uso:

```
test -f /home/alumno && echo "Existe directorio"
```

La salida sera:

```
Existe Directorio
```

En la forma resumida se puede escribir:

```
[ -f /home/albertjh/googleearth ] && echo "fichero existe…"
fichero existe…
```

Como vemos es lo mismo.

Expresiones test

Estas son algunas de las más comunes:

Comprobación directorios:

-f /ruta/nombre –> Comprueba si es un fichero normal
-l /ruta/nombre –> Comprueba si es un enlace suave
-d /ruta/ –> Comprueba que existe el directorio
-x /ruta/nombre –> Comprueba si es un ejecutable
-u /ruta/nombre –> Comprueba si tiene activados los permisos suid
-g /ruta/nombre –> Comprueba si tiene activados los permisos sgid
-s /ruta/nombre –> comprueba que su tamaño es mayor a 0

Comprobación de cadenas:

"cadena" = "cadena2" –> Comprueba si son iguales
-z cadena –> Comprueba si está vacía
"cadena" != "cadena2" –> Comprueba que son diferentes

Comprobación de expresiones numéricas:

exp **-eq** exp2 –> Comprueba si son iguales
exp **-ge** exp2 –> Comprueba si exp >= exp2
exp **-ne** exp2 –> Comprueba si exp distinto de exp2
exp **-gt** exp2 –> Comprueba si exp > exp2
exp **-le** exp2 –> Comprueba si exp <= exp2
exp **-lt** exp2 –> Comprueba si exp < exp2

Para concatenar expresiones a evaluar:

-o = OR
-a = AND
! = NOT

Algunos ejemplos del uso de test:

numeros.sh

```
#! /bin/bash
clear
read -d “,” -p “Introduce dos números separados por comas ” num1 num2
if [ -z $num1 -o -z $num2 ]
then
echo “Debes introducir dos números, por favor”
elif [ $num1 -eq $num2 ]
then
echo “Los números son iguales”
elif [ $num1 -gt $num2 ]
then
echo “El $num1 > que $num2”
fi
```

BuscaFich.sh

```
#! /bin/bash
clear
read -p “Introduce directorio a buscar… ” direct
read -p “Nombre de fichero a buscar… ” nombre
if [ ! -d "direct ]
then
echo “$direct no existe”
else
find $direct -name “*.$nombre” -exec ls -l ‘{}’ \;
fi
```

5.2- While

El ***while*** se estructura de la siguiente forma:

```
while condicion
do
break
done
```

While se usa para repetir un conjunto de comandos/instrucciones dependiendo de si se cumple o no la condición. La condiciones que se pueden poner en el while son con el comando test , poniendo un true (poniendo un true en el ***while*** se crea un bucle infinito) o poner un comando con comillas invertidas. El ***break*** se pone solo si quieres salir bruscamente del bucle. Veamos tres ejemplos(el primero se podría haber echo con un ***select***, por si alguien quiere probarlo):

Calculadora.sh

```
#! /bin/bash
clear
opcion=2
while [ $opcion -ne 5 ]
do
echo "1.suma"
echo "2.resta"
echo "3.multiplicación"
echo "4.división"
```

```
echo "5.salir"
read -n 1 -p "Introduce una opcion " opcion
case $opcion in
1) read -p "Introduce el 1 numero " numero1
read -p "Introduce el 2 numero " numero2
echo "El resultado es `expr $numero1 + $numero2`" ;;
2) read -p "Introduce el 1 numero " numero1
read -p "Introduce el 2 numero " numero2
echo "El resultado es `expr $numero1 - $numero2`" ;;
3) read -p "Introduce el 1 numero " numero1
read -p "Introduce el 2 numero " numero2
echo "El resultado es `expr $numero1 '*' $numero2`" ;;
4) read -p "Introduce el 1 numero " numero1
read -p "Introduce el 2 numero " numero2
echo "El resultado es `expr $numero1 '/' $numero2`" ;;
esac
done
```

Este script funciona como una calculadora, posee 5 opciones (suma, resta, multiplicación, división y salir), nos interesa que podamos realizar tantas veces como queramos cualquier cálculo hasta que pulsemos un 5 por lo tanto es necesario una estructura repetitiva. El 5 es para salir por lo tanto el bucle se repetirá hasta que la variable ***opcion*** valga 5 (es decir, cuando introduzcas por teclado un 5) por eso hay que usar el comando ***test*** ([$opcion -ne 5]) , cuando eso ocurra ya no se cumple la condición del ***while*** y el programa finaliza.

Si en el ***while*** vas a poner una variable, debe declararse antes ya que sino no entra, darle cualquier valor. Si no se le da valor antes, la variable no valdrá nada (en el ejemplo anterior hubiera fallado ya que estás diciendo que se hace el bucle mientras ” ” no sea igual a 5, sin embargo al dar valor a la variable opción antes del ***while***, entra)

BucleInfinito.sh

```
#! /bin/bash
clear
while true
do
read -p "Introduce la palabra fin para salir del bucle " fin
if [ "$fin" = "fin" ]
then
exit 2;
fi
done
```

Este script es un ejemplo de como hacer un bucle infinito, hasta que no escribas un “***fin***“, no saldrá del programa.

Con el ***while*** se puede leer línea por línea un fichero, para ello lo único que hay que hacer es poner un redireccionamiento de entrada en el ***done*** con la ruta del fichero que queremos leer (esto es muy útil, ya que puedes crear usuarios a partir de un fichero, etc. Tan solo tenéis que poner `done < /ruta_fichero`).

LecturaFichero.sh

```
#! /bin/bash
clear
numLinea=0
linea="nada"
while [ ! -z "$linea" ]
do
read linea
if [ ! -z "$linea" ]
then
numLinea=`expr $numLinea + 1`
echo "$numLinea. $linea"
fi
done < /etc/passwd
```

La función de este script es enumerar las lineas del fichero /etc/passwd, leerá línea por línea hasta que encuentre una línea que esté vacía, cuando la encuentre dejará de leer el fichero, a pesar de que haya o no más líneas después de la línea en blanco (aunque como es el fichero de los usuario registrados en el sistema, la única línea en blanco que habrá será la última del fichero). Se debe tener en cuenta que antes del bucle se ha inicializado la variable línea.

5.3- Until

La estructura repetitiva ***until*** es de la siguiente forma:

```
until condicion
do
break
done
```

La estructura ***until*** se usa para repetir un conjunto de comando hasta que se cumpla la condición, cuando se cumple el script sale del ***until***. Las condiciones y el ***break*** es lo mismo que en el ***while***, si se usa una variable en el ***until*** se debe declarar antes.

Ejemplo:

BorrarFicheros.sh

```
#! /bin/bash
clear
directorio=malo
until `cd $directorio 2> /dev/null`
do
clear
read -p "Introduce un directorio " directorio
done
echo "borrado fichero"
rm -i $directorio/*
```

Este script comprueba si el directorio existe, si el directorio que introduces no existe te volverá a pedir la ruta, una vez introduzcas una ruta que existe, entonces saldrá del ***until*** y borrará todos los ficheros de su interior

6- PARÁMETROS POSICIONALES

¿Qué es un parámetro posicional?

Son valores que se le pasan al script desde la línea de comandos cuando se ejecuta, se numeran en orden del 1 al 12.
Pero a partir del 10 hay que encerrarlo entre llaves, ej. *${11}*

```
$ ./Script valor1 valor2 … valorN
```

./Script = valor0

Para ver el contenido de las variables se utiliza el **$NumParámetro**
El conjunto de todos los parámetros se puede recuperar de golpe con **$***
El número de parámetros que se le pasan al script esta definido como **$#**

BorrarFicheroParametros.sh

```
#! /bin/bash
# Al ejecutar como 1° parámetro un directorio, 2° como fichero a
borrar
clear
if [ $# -ne 2 ]
then
echo "Debes ejecutarlo así: $0 directorio nombreFichero"; exit 65
elif [ ! -d $1 ]
then
echo "El parámetro 1 no es un directorio!! "; exit 65
elif [ ! -f $1/$2 ]
then
echo "El parámetro 2 no es un fichero!; exit 65
else
echo "Borrando el fichero… "
rm -fi $1/$2
fi
```

Vamos comprobando poco a poco si es un directorio correcto, si lo es, pasamos a comprobar el fichero, y si lo es lo borramos.

BorrarFicheroParametrosBucle.sh

```
#! /bin/bash
clear
[ ! -d $1 ] && (echo "Directorio no existe"; exit 65) || exit 65
for fichero in `ls $1`
do
[ -f $fichero ] && rm -f &fichero
done
```

Shift: Este comando desplaza elementos a la izquierda machacando el primero y se pierde, un ejemplo:

ParametrosBucleShift.sh

```
#! /bin/bash
clear
while [ "$1" != "" ]
do
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} ${12}"
shift
done
```

El resultado es curioso, sería el siguiente:

```
./ParametrosBucleShift.sh 1 dos tres 4 5 seis siete 8 nueve 10 11
doce

1 dos tres 4 5 seis siete 8 nueve 10 11 doce
dos tres 4 5 seis siete 8 nueve 10 11 doce
tres 4 5 seis siete 8 nueve 10 11 doce
4 5 seis siete 8 nueve 10 11 doce
5 seis siete 8 nueve 10 11 doce
seis siete 8 nueve 10 11 doce
siete 8 nueve 10 11 doce
8 nueve 10 11 doce
nueve 10 11 doce
10 11 doce
11 doce
doce
```

Por último me queda hablar de *set*, aunque no tiene mucha utilidad a esta función.

```
set [ -$variable ] [`comando`]
```

Si aparece cambia el valor de los parámetros posicionales o por el contenido de una variable o por el resultado de la ejecución de un comando.

7- IFS Delimitador

IFS= delimitador (**I**nput **F**ile **S**eparator)

Sirve para cambiar el delimitador por defecto, que es " " , así puedes evitar decir:

read -d ";"

Pero ojo, en todo el script se usará como separador un ";" si tu IFS=; así que cuidado

8- ARRAYS

Las ***arrays*** de los script funcionan de la misma forma que los ***arrays*** de cualquier lenguaje de programación. Una ***array*** es un conjunto o agrupación de valores cuyo acceso se realiza por índices, en un script se puede almacenar en un mismo ***array*** todo tipo de cosas, números, cadenas, caracteres, etc.

En las ***arrays*** el primer elemento que se almacena lo hace en la posición 0 (en el ejemplo seria Paco). En los script no hace falta declarar el tamaño de la ***array***, puedes insertar tantos valores como desees. Para declarar una ***array*** es:

```
declare -a nombre_array
declare -a nombres
```

La opción -a sirve para decir que lo que vas a declarar es una ***array***.

Para darle valores se puede hacer de dos formas:

1. Darle valores posición por posición.

```
nombre_array[posicion]=valor
nombres[3]=Manolo
```

2.Darle todos los valores de golpe (aunque también se puede decir la posición deseada en la que quieres guardar uno de los valores).

```
nombre_array=( valor1 valor2 valor3 [posicion]=valor4 ..... valorN )
nombres=( Maria Alberto Rodrigo [7]=Paco )
```

Para ver el contenido de la ***array*** en una posición:

```
${nombre_array[posicion]}
${nombres[0]}
```

Para saber cuantos elementos contiene la ***array***:

```
${#nombre_array[*]}
${#nombres[*]}
```

Para recuperar todos los elementos de una ***array***:

```
${nombre_array[*]}
${nombres[*]}
```

A continuación un ejemplo de arrays:

arrays.sh

```
#! /bin/bash
clear
contador=0
declare -a usuario=( Alberto John Roberto Laura Sergio Cristian
Dani )
for valor in ${usuario[*]}
do
echo "El usuario $contador vale $valor"
contador=`expr $contador + 1`
done
```

9- FUNCIONES

En el ámbito de la **programación**, una **función** es un tipo **subalgoritmo**, es el término para describir una secuencia de órdenes que hacen una tarea específica de una **aplicación** más grande.

Es la forma en la que más me gusta programar, modulando todo en partes pequeñas para después obtener una mayor, con lo cual el programar algo se hace fácil y divertido.

```
function nombreFuncion (){
comando1
comando2
…
[ return codigoSalida ]
}
```

También se especifica sin poner function, pero puede llegar a dar problemas así que se recomienda ponerlo.

El código de salida especificado por un return es el código de salida del resultado de la ejecución de todos los comandos en la función. Si no se especifica un return devolverá el de la última salida de esa función.

Dentro de una función se pueden definir variables locales (solo reconocidas por esa función) y se especifican así:

```
local nombreVariable
```

Importante, las funciones se declaran al principio de los scripts!

sumaFuncion.sh

```
#! /bin/bash
clear
function suma() {
local resultado
read -p "Introduce el primer numero: " num1
read -p "Introduce el segundo numero: " num2
resultado=`expr $num1 + $num2`
return $resultado
}
#---Cuerpo del script---
echo "Llamo a la funcion suma"
suma
echo "El resultado es $?"
```

En el paso de parámetros en una función no se pueden definir como en otros lenguajes las variables que le pasas dentro de los paréntesis, sino que se pasan los valores poniéndolos a continuación del nombre de la función:

```
nombreFuncion valor1 valor2 valor 3 …
```

Dentro de una función esos valores se recogen como:

valor1=**$1** valor2=**$2** …

sumaFuncionParametros.sh

```
#! /bin/bash
clear
function suma() {
local resultado
resultado=`expr $1 + $2`
return $resultado
}
#--Cuerpo del script--
read -p "Introduce el primer numero: " num1
read -p "Introduce el segundo numero: " num2
echo "Llamo a la funcion suma"
suma $num1 $num2
echo "El resultado es $?"
```

10- LEER UN FICHERO

Para leer un fichero es necesario un bucle en la primera línea mas un **EOF** (End Of File) y redireccionar la entrada para ese bucle:

```
while condición
do
read linea
comandos…
done < /ruta/fichero
```

Esto se verá mejor con un ejemplo:

Tenemos el siguiente fichero en */tmp/ejemplo_texto* y contiene:

En un Lugar de la Mancha
de cuyo nombre no quiero acordarme
y mucho mas….

Vamos a hacer un pequeño script que nos cuente el número de líneas que tiene el fichero, como lo hace **wc**:

LeerFichero.sh

```
#! /bin/bash
clear
linea="algo"
while [ ! -z "$linea" ]
do
read linea
num_linea=`expr $num_linea + 1`
if [ ! -z "$linea" ]
then
echo "La linea numero: $num_linea del fichero es… $linea"
fi
done < /tmp/ejemplo_texto
echo "Total lineas: `expr $num_linea - 1`"
```

Y como resultado tendríamos:

```
$ ./LeerFichero.sh
La linea numero: 1 del fichero es… En un Lugar de la Mancha
La linea numero: 2 del fichero es… de cuyo nombre no quiero
acordarme
La linea numero: 3 del fichero es… y mucho mas.
Total lineas: 3
```

11- CREACIÓN DE DEMONIOS

La verdad es que el echo de habilitar un ***demonio*** no tiene nada que ver con hacer un script, pero es muy útil saber como lanzarlo como ***demonio***.

Un proceso ***demonio*** es un proceso que lanza INIT (el padre de los procesos) al iniciar el pc en un determinado ***runlevel*** (es el nivel de ejecución del sistema) y lo lanza en segundo plano o BACKGROUND y esta siempre atento a la llamada del usuario (el usuario puede lanzar ese proceso cuando quiera y tantas veces quiera, para entenderlo mejor, son programas que al iniciarse el pc se ejecutan enseguida).

E n linux hay 6 niveles de ejecución del sistema, y según sea el nivel *init* lanza unos determinados programas que se encuentran en /etc/rcnúmero_ejecucion.d (Ej: /etc/rc5.d) Los programas que hay dentro se llaman de la siguiente forma:

```
SNúmeroNombrePrograma -----> S01AreaSwap
```

La **S** significa START, son los que INIT lanzará y el número es la prioridad con que los lanzará. Si el número es muy pequeño, será de los primeros que se lancen, y si es muy grande, es de los últimos.

Los niveles que hay son:

0 ---> Apagado sistema

1 ---> Monousuario, sin entorno gráfico, sin soporte de red

2 ---> Multiusuario, sin entorno gráfico, sin soporte de red

3 ---> Multiusuario, sin entorno gráfico, con soporte de red

4 ---> RESERVADO

5 ---> Multiusuario, con entorno gráfico, con soporte de red (se lanza por defecto)

6 ---> Reinicia sistema

Los pasos para crear un demonio son:

1.Situarse en /etc/init.d y copiar en esa situación el archivo que haremos demonio.

```
cd                                                    /etc/init.d
cp /CreaDemonios.sh .
```

2. Situarte en /etc/rc5.d (no hay mucha gente que inicie el PC sin entorno gráfico) y crear un enlace suave ("soft link") con el nombre SnumeroPrioridadNombrePrograma, pero la prioridad debe ser una que no no esté repetida.

```
cd                                                    /etc/rc5.d
ln -s /etc/init.d/CreaDemonios.sh ./S09CreaDemonios
```

3. Por último reiniciamos para aplicar los cambios.

```
init 6
```

El siguiente ejemplo ejecuta todos estos pasos

```
1.#! /bin/bash
2.clear
3.contador=0
4.declare -a arrayPrioridades
5.usuario=`whoami`
6.if [ "$usuario" = "root" ]
7.then
8.     if [ -f /etc/rc5.d/README ]
9.     then
10.         rm -f /etc/rc5.d/README
11.     fi
12.
13.read -p "Introduce la ruta del archivo: " archivo
14.if [ ! -f $archivo ]
15.then
16.     echo "El fichero $archivo no existe"
17.     exit 3
18.else
19.     nombreArchivo=`basename $archivo|cut -d "." -f 1`
20.     cd /etc/init.d
21.     cp $archivo .
22.     chmod 700 ./`basename $archivo`
23.     cd /etc/rc5.d
24.     for ficheros in `ls .`
25.     do
26.         nombreFichero=`basename $ficheros`
27.         prioridad=`expr substr $ficheros 2 2`
```

```
28.         arrayPrioridades[$contador]=$prioridad
29.         contador=`expr $contador + 1`
30.     done
31.     contador=1
32.     for interiorArray in ${arrayPrioridades[*]}
33.     do
34.         if [ "$contador" != "${#arrayPrioridades[*]}" ]
35.         then
36.               condicion=`expr   ${arrayPrioridades[$contador]}   -
$interiorArray`
37.             if [ $condicion -gt 1 ]
38.             then
39.                 if [ $interiorArray -lt 10 ]
40.                 then
41.                     prioridad=`expr $interiorArray + 1`
42.                           ln  -s  /etc/init.d/`basename  $archivo`
./S0$prioridad$nombreArchivo  &&  echo  "Se  ha  creado  el  demonio
correctamente"
43.                     chmod 700 ./S0$prioridad$nombreArchivo
44.                     creado="DEMONIO"
45.                     break
46.                 else
47.                     prioridad=`expr $interiorArray + 1`
48.                           ln  -s  /etc/init.d/`basename  $archivo`
./S$prioridad$nombreArchivo  &&  echo  "Se  ha  creado  el  demonio
correctamente"
49.                     chmod 700 ./S0$prioridad$nombreArchivo
50.                     creado="DEMONIO"
51.                     break
52.                 fi
53.             fi
54.
55.         fi
56.         contador=`expr $contador + 1`
57.     done
58.
59.
60.fi
61.if [ "$creado" = "DEMONIO" ]
62.then
63.     read -n 1 -p "¿Desea reiniciar el ordenador para aplicar los
cambios <s/n>? " respuesta
64.     if [ "$respuesta" = "s" ]
65.     then
66.         init 6
67.     fi
68.fi
```

```
69.else
70.     echo "No eres el root,no puedes ejecutar este script"
71.fi
```

Otra forma de crear un demonio:

Esta vez solo se necesitan un par de líneas.

1.Tenemos nuestro script preparado y funcionado, en este ejemplo lo llamaremos *demonio.sh*
2.Le damos permisos 755, `sudo chmod 755 demonio.sh`
3.Lo movemos a */etc/init.d*
4.Por último lo convertimos `sudo update-rc.d demonio.sh defaults`

Esto es todo, más simple pero de la otra forma se puede ver lo que realmente hace este comando.

update-rc.d actualiza automáticamente los enlaces a los scripts de init tipo System V que se encuentran en **/etc/rc[nivel_de_ejecución].d/NNnombre** y que apuntan a los script **etc/init.d/nombre**.

Cuando se ejecuta con una o varias de las opciones **defaults, start,** o **stop**, *update-rc.d* crea los enlaces /**etc/rc[nivel_de_ejecución].d/[SK]NNnombre** apuntando al script */etc/init.d/nombre*.

Si se usa la opción **defaults** entonces *update-rc.d* creará enlaces para arrancar los servicios en los niveles de ejecución **2345** y parar los servicios en los niveles de ejecución **016**.En vez de usar *defaults*, los niveles de ejecución en los que se arranca o se para un servicio pueden ser especificados explícitamente mediante un conjunto de argumentos:

Cada uno de estos conjuntos empieza con un argumento **start** o **stop** para especificar cuándo se van a crear enlaces para arrancar o parar el servicio.

11.1- Ejecutar un demonio o script sin contraseña

Muchas veces necesitamos ejecutar un script en el cual se requiere cambiar de usuario o ejecutarlo con derechos de sudo. Para ello, haremos lo siguiente:

En el /etc/sudoers

usuario ALL = NOPASSWD: /usr/local/bin/script1,

Después, el usuario puede hacer sudo /usr/local/bin/script1 y va a correr como root.

12- USO DE CRON

Digamos que el cron se compone básicamente de dos "partes" el daemon y el fichero de configuración. El daemon se llama crond y es el encargado de leer el fichero de configuración /etc/crontab.

Lo hace cada 60 segundos, en busca de cambios en dicho fichero e incorporar así tareas al sistema.

Nosotros no vamos a entrar en el deamon, ya que bastará que lo marqueis para que se ejecute al iniciar vuestro sistema, esto lo podeis hacer con la herramienta que traiga vuestra distribución sino, siempre podeis recurrir al modo texto con: *chkconfig*.

Bien, una vez editado dicho fichero encontraremos esto:

```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

El primer campo, como bien veis, es el entorno donde se ejecutarán las órdenes, podeis dejarlo así por defecto, a menos que useis otra shell.

El segundo campo está claro, es el valor de la variable PATH, si las ordenes que ejecutará la tarea no se encuentran en dicho PATH, no teneis más que añadirlo.

El tercero, manda los outputs del cron al root, o a quien queramos.

Si queremos que no se envie nada sólo debemos dejar la variable así: *MAILTO=""*.
Los /etc/cron.hourly, daily...son fichero que usa cron para su gestión interna y los lanza cada día.

Bien, ahora queremos agregar una tarea a nuestro sistema, ¿cómo se hace?. Debemos saber cómo funcionan los campos del /etc/crontab.

El primer campo son los minutos.

El segundo campo son las horas.
El tercer campo es el día.
El cuarto es el día de la semana.
El quinto es la orden a ejecutar.
Es decir, para el crontab el fichero es así:

minute hour day month dayofweek command

Hay que destacar que los minutos deben ser enteros de entre 0-59.

Las horas, enteros de 0-23.

Días, 1-31. (Para meses que los tengan, claro.)

Meses, 1-12, también se pueden usar nombres cortos, (en inglés) jan, feb...(se aconseja usar números).

Días de la semana, se usan enteros 1-7 (al igual que en los meses, se puede usar el nombre corto, pero tambien lo desaconsejamos, en favor de los números.)

Y por último la orden, por ejemplo puede ser: *ls -h > /home/manssson/ls.txt*.

Con esto ya deberíamos ser capaces de crear y hacer funcionar cualquier tarea, pero cron tiene alguna más utilidades que conviene saber.

Por ejemplo, si ponemos un asterisco (*) en algún campo, por ejemplo el mes, esto hará que se ejecute la orden todos los meses, y luego respetará los demás valores, tales como el día o la hora.

Si queremos indicar un rango de enteros, lo haremos con un guión (-). Si queremos indicar enteros separados, se harán con comas (,), por ejemplo: *1, 2, 3.*

Podemos concretar o excluir valores usan la barra (/), es decir, si queremos que una orden se ejecute durante los 0-59 minutos de una hora menos en el minuto 15, pues lo marcaremos haciendo:0-59/15.

También podemos hacer que una orden que se ejecute en un determinado minuto, haciendo:*/25. Esto hará que nuestra tarea se ejecute siempre en el minuto 25. El uso de la barra (/) y el asterisco (*) puede emplearse en todos los campos. Si queremos que una tearea no sea ejecutada sólo debemos comentar esa linea al principio con la "almohadilla" (#): #30 18 8 * * root rm -f /var/cache/apt/archives/*.rpm.

Otra opción es copiar el script en alguna de las carpetas por defecto cron.*

Por ejemplo, si queremos una periodicidad diaria, lo copiaremos en /etc/cron.daily

Con todo esto ya podremos incorporar tareas a nuestro sistema de forma automática.

13- COLORES EN LA CONSOLA

Tenemos varios tipos de colores, se distinguen por *background* (el fondo) y *foreground* (las letras). Esto se puede hacer metiendolo en variables o a mano

Según la **secuencia de escape ANSI**, podemos cambiar el color del *background* con \033[x;**4**xm y el *foreground* con \033[x;**3**xm

\033 indica la marca de escape ANSI y lo siguiente es la secuencia de colores.

La lista de los posibles colores varía según la terminal y el entorno gráfico, pero por defecto son:

Negro	0;30	Gris oscuro	1;30
Azul	0;34	Azul claro	1;34
Verde	0;32	Verde claro	1;32
Cyan	0;36	Cyan claro	1;36
Rojo	0;31	Rojo claro	1;31
Purpura	0;35	Purpura claro	1;35
Marron	0;33	Amarillo	1;33
Gris claro 0;37	blanco 1;37		

Los colores más intensos empiezan por cero: **[0-9]**;xx , al igual que los colores que son x;x**[0-9]** esto solo es una pequeña demostración de colores. Sin embargo la cadena de intensidad de colores tiene una pequeña curiosidad:

0 es intenso
1 y 6 es normal
2 y 8 es color de fondo de la bash (oculto)
3 negativo
4 inverso
5 parpadeante
7 intercambio de foreground por background

Pero en consolas gráficas:

1 y 4 subrayado
6 tachado
7 inverso
8 oculto

Existe el problema de que cuando se pone un color, al terminar la secuencia todo lo demás se queda de ese color si no se sustituye por otro, para que esto no ocurra podemos recurrir a poner al final del todo un: **\033[0m** así quedará el color de la consola inicial.

Así pues sabiendo esta tabla de valores, podemos escribir un par de líneas:

```
$ PS1="\[\033[1;34m\][\u@\h:\w]$\[\033[0m\] "
[usuario@sunny02:~]$
```

Como puedes ver se ha cambiado el prompt por el color azul. Pero también se puede combinar con el fondo de esta manera:

```
$ PS1="\[\033[44;1;31m\][\u@\h:\w]$\[\033[0m\] "
```

De esta forma podemos poner las letras rojas y el fondo azul. Utilizando esto, podemos crear scripts con colores y que queden muy vistosos.

Recordar que para hacer *echo's* poniendo las secuencias de escape, deberéis utilizar la opción **-e**

Un ejemplo muy curioso, que combina colores con un *sleep*, para hacer creer al usuario que está cargado algo y que sale ok:

```
 clear; echo "CARGANDO, ESPERE POR FAVOR";echo "";for (( j=1; j<=10;
j++ )); do echo -e "EJEMPLO $j \c" ;for (( i=0; i<=30; i++ ));do echo
-e "\033[0;34m.\c";sleep 0.05;done; echo -e "\c";echo -e "\t\033[0;32m
[OK]\033[0m";done; echo "\033[0;32mCARGADO CON EXITO\033[0m"

CARGANDO, ESPERE POR FAVOR

EJEMPLO 1 ..... [OK]
EJEMPLO 2 ..... [OK]
EJEMPLO 3 ..... [OK]
EJEMPLO 4 ..... [OK]
EJEMPLO 5 ..... [OK]
EJEMPLO 6 ..... [OK]
EJEMPLO 7 ..... [OK]
EJEMPLO 8 ..... [OK]
EJEMPLO 9 ..... [OK]
EJEMPLO 10 ..... [OK]
CARGADO CON ÉXITO
```

Nota: Algunos de los ejemplos de este documentos han sido extraidos de "Diario de un Linuxero"


Aula
Sun
UCM
Sun