

PRÁCTICA 3 - CSO

1. ¿Qué es el Shell Scripting? ¿A qué tipos de tareas están orientados los scripts? ¿Los scripts deben compilarse? ¿Por qué?

El Shell Scripting es la automatización de tareas mediante la escritura de secuencias de comandos que pueden ser ejecutadas por la shell del sistema operativo. Estos scripts pueden contener una serie de comandos, estructuras de control, variables y funciones para realizar diversas tareas de manera más eficiente y automatizada.

Los scripts *pueden estar orientados a una amplia variedad de tareas*, incluyendo, pero no limitándose a

- Automatización de tareas repetitivas, como la copia de archivos, la descarga de datos, la organización de archivos, etc.
- Administración del sistema, como la creación y gestión de usuarios, la programación de tareas, el monitoreo de recursos, etc.
- Procesamiento de datos, como la manipulación y análisis de archivos de texto, CSV, JSON, etc.
- Configuración y personalización del entorno de usuario.
- Creación de secuencias de comandos para despliegues de software y configuración de servidores.
-

En el Shell Scripting, los scripts **no se compilan en el sentido tradicional** como sucede con lenguajes de programación compilados, como C o C++. En su lugar, los scripts se escriben en un lenguaje de secuencias de comandos (como Bash, Python, Perl, etc.) y son interpretados directamente por la shell del sistema operativo. Esto significa que no es necesario compilarlos antes de su ejecución.

La principal ventaja de los scripts interpretados es su flexibilidad y facilidad de desarrollo, ya que los cambios en el código se reflejan inmediatamente. Sin embargo, esto también puede hacer que los scripts sean más lentos en comparación con programas compilados, ya que cada línea de código se interpreta en tiempo de ejecución en lugar de ejecutarse directamente en código de máquina.

En resumen, el Shell Scripting es una forma poderosa de automatizar tareas en sistemas operativos Unix y Linux, sin necesidad de compilar los scripts, ya que son ejecutados directamente por la shell, lo que facilita la escritura, modificación y ejecución rápida de tareas automatizadas.

2. Investigar la funcionalidad de los comandos echo y read

- Comando **echo** → imprime mensajes

```
dolores@dolores-VirtualBox:~$ echo "Ingrese nombre y apellido: "  
Ingrese nombre y apellido:  
dolores@dolores-VirtualBox:~$
```

- Comando **read** → lee variables

```
dolores@dolores-VirtualBox:~$ read nombre apellido  
dolores garro  
dolores@dolores-VirtualBox:~$ echo "Su nombre y apellido es: $nombre $apellido "  
Su nombre y apellido es: dolores garro
```

a) ¿Cómo se indican los comentarios dentro de un script?

Los comentarios se indican con #.

b) ¿Cómo se declaran y se hace referencia a variables dentro de un script?

Declaración:

NOMBRE="dolores" → SIN ESPACIOS entre el igual!!!

Acceso al contenido: con el signo \$.

\$NOMBRE

3. Crear dentro del directorio personal del usuario logueado un directorio llamado practicashell-script y dentro de él un archivo llamado mostrar.sh cuyo contenido sea el siguiente:

```
#!/bin/bash  
# Comentarios acerca de lo que hace el script  
# Siempre comento mis scripts, si no hoy lo hago  
# y mañana ya no me acuerdo de lo que quise hacer  
echo "Introduzca su nombre y apellido:"  
read nombre apellido  
echo "Fecha y hora actual:"  
date  
echo "Su apellido y nombre es:"  
echo "$apellido $nombre"  
echo "Su usuario es: `whoami`"  
echo "Su directorio actual es:"
```

- a) Asignar al archivo creado los permisos necesarios de manera que pueda ejecutarlo
- b) Ejecutar el archivo creado de la siguiente manera: `./mostrar`
- c) ¿Qué resultado visualiza?

```
/home/dolores/practica-shell-script$ ./mostrar.sh
Introduzca su nombre y apellido
dolores garro
Fecha y hora actual
vie 22 sep 2023 15:51:39 -03
Su apellido y nombre es:
garro dolores
Su usuario es dolores
Su directorio actual es
/home/dolores/practica-shell-script
```

- d) Las backquotes (```) entre el comando `whoami` ilustran el uso de la sustitución de comandos. ¿Qué significa esto?

La backquote (```) se utiliza en la programación de shell para realizar la sustitución de comandos, lo que significa que el comando entre las comillas se ejecuta y su resultado se utiliza en su lugar en el contexto más amplio del script.

`whoami`, se ejecuta como un comando independiente.

El resultado de `whoami`, que es el nombre de usuario actual, se toma y se coloca en su lugar en la línea de comando original

- e) Realizar modificaciones al script anteriormente creado de manera de poder mostrar distintos resultados (cuál es su directorio personal, el contenido de un directorio en particular, el espacio libre en disco, etc.). Pida que se introduzcan por teclado (entrada estándar) otros datos.

Para crear dentro del directorio personal un directorio que se llame `practicashell-script` y dentro de él un archivo llamado `mostrar.sh` primero

`cd` → para asegurarme que estoy dentro del directorio personal
`mkdir practica-shell.script` → para crear el directorio
`cd practica-shell.script` → para posicionarme en ese directorio
`vim mostrar.sh` → para crear el archivo
escribir en el archivo lo pedido

4. Parametrización: ¿Cómo se acceden a los parámetros enviados al script al momento de su invocación? ¿Qué información contienen las variables \$#, \$*, \$? Y \$HOME dentro de un script?

\$# → almacena el número de parámetros o parámetros que se pasaron al script.

\$* → almacena los argumentos pasados al script como una sola cadena.

```
for arg in "$*"; do
    echo "Argumento: $arg"
done
```

\$ → almacena el código de retorno del último comando ejecutado en el script. Si el retorno es 0 es porque el código se ejecutó correctamente.

\$HOME → almacena la ruta al directorio personal del usuario que ejecutó el script

```
echo "Mi directorio personal es: $HOME"
```

5. ¿Cual es la funcionalidad de comando exit? ¿Qué valores recibe como parámetro y cuál es su significado?

Finaliza la ejecución del script. Ningún código después del comando exit se ejecutará.

Los valores de retorno y sus significados son:

- exit ó exit 0: finalización exitosa del script.
- exit *n*: el script se detuvo debido a un error. (*n* puede ser cualquier número entero en el rango de 1 a 255).
-

6. El comando expr permite la evaluación de expresiones. Su sintaxis es: expr arg1 op arg2, donde arg1 y arg2 representan argumentos y op la operación de la expresión. Investigar que tipo de operaciones se pueden utilizar.

Operaciones Aritméticas:

- Suma: ``arg1 + arg2``
- Resta: ``arg1 - arg2``
- Multiplicación: ``arg1 * arg2`` (nótese que el asterisco `*` debe ser escapado con `\`)
- División: ``arg1 / arg2``
- Módulo (resto de la división): ``arg1 % arg2``

Operaciones de Comparación:

- Igual: ``arg1 = arg2``
- No igual: ``arg1 != arg2``
- Mayor que: ``arg1 > arg2``
- Menor que: ``arg1 < arg2``
- Mayor o igual que: ``arg1 >= arg2``
- Menor o igual que: ``arg1 <= arg2``

Operaciones Lógicas:

- AND lógico: ``arg1 & arg2``
- OR lógico: ``arg1 | arg2``
- NOT lógico: ``! arg1``

Operaciones de Concatenación de Cadenas:

- Concatenar cadenas: ``arg1 . arg2``

Operadores de Asignación:

- Asignación de valor: ``arg1 = arg2``

Operador de Longitud de Cadena:

- Longitud de cadena: ``length arg1``

7. El comando “test expresión” permite evaluar expresiones y generar un valor de retorno, true o false.

Este comando puede ser reemplazado por el uso de corchetes de la siguiente manera `[expresión]`. Investigar qué tipo de expresiones pueden ser usadas con el comando test. Tenga en cuenta operaciones para: evaluación de archivos, evaluación de cadenas de caracteres y evaluaciones numéricas.

- Comprueba si el archivo existe:

```
test -e archivo  
[ -e archivo ]
```

- Comprueba si el archivo es un directorio:

```
test -d directorio  
[ -d directorio ]
```

- Comprueba si un archivo es un enlace simbólico:

```
test -h archivo  
[ -h archivo ]
```

- Comprueba si un archivo tiene permisos de lectura escritura o ejecución

```
test -r archivo  
[ -r archivo ]  
test -w archivo  
[ -w archivo ]  
test -x archivo  
[ -x archivo ]
```

- Comprueba si dos archivos son iguales

```
test archivo1 -ef archivo2  
[ archivo1 -ef archivo2 ]
```

Evaluación de cadenas de caracteres

1. Comprobar si una cadena no está vacía:

```
bash Copy code  
  
test -n cadena  
[ -n cadena ]
```

2. Comprobar si una cadena está vacía:

```
bash Copy code  
  
test -z cadena  
[ -z cadena ]
```

3. Comprobar si dos cadenas son iguales:

```
bash Copy code  
  
test cadena1 = cadena2  
[ cadena1 = cadena2 ]
```

4. Comprobar si dos cadenas no son iguales:

```
yaml Copy code  
  
test cadena1 != cadena2  
[ cadena1 != cadena2 ]
```

Evaluaciones numéricas

1. Comprobar si dos números son iguales:

```
bash Copy code  
  
test num1 -eq num2  
[ num1 -eq num2 ]
```

2. Comprobar si dos números no son iguales:

```
perl Copy code  
  
test num1 -ne num2  
[ num1 -ne num2 ]
```

3. Comprobar si un número es menor que otro:

```
perl Copy code  
  
test num1 -lt num2  
[ num1 -lt num2 ]
```

4. Comprobar si un número es menor o igual que otro:

```
bash Copy code  
  
test num1 -le num2  
[ num1 -le num2 ]
```

5. Comprobar si un número es mayor que otro:

```
perl Copy code  
  
test num1 -gt num2  
[ num1 -gt num2 ]
```

6. Comprobar si un número es mayor o igual que otro:

```
bash Copy code  
  
test num1 -ge num2  
[ num1 -ge num2 ]
```

8. Estructuras de control. Investigue la sintaxis de las siguientes estructuras de control incluidas en shell scripting:

- if
- case
- while
- for
- select

| IF | CASE |
|--|--|
| <pre> if [condición] then # Código si la condición es verdadera elif [otra_condición] then # Código si la otra condición es verdadera else # Código si ninguna de las condiciones es verdadera fi </pre> | <pre> case \$variable in v1) # Código si la variable es igual a v1 ;; v2) # Código si la variable es igual a v2 ;; *) # Código si la variable no coincide con ningún valor anterior ;; esac </pre> |
| WHILE | FOR |
| <pre> while [condición] do # Código que se ejecuta mientras la condición sea verdadera done </pre> | <pre> for variable in valor1 valor2 valor3 do # Código que utiliza la variable con los valores especificados done </pre> |

SELECT

```
select variable in opcion1 opcion2 opcion3
do
  case $variable in
    opcion1)
      # Código para la opción 1
      ;;
    opcion2)
      # Código para la opción 2
      ;;
    opcion3)
      # Código para la opción 3
      ;;
    *)
      # Código si se selecciona una opción no válida
      ;;
  esac
done
```

9. ¿Qué acciones realizan las sentencias **break** y **continue** dentro de un bucle? ¿Qué parámetros reciben?

- **break [n]** → corta la ejecución de *n* niveles de loops
- **continue [n]** → salta la iteración actual de un bucle y continúa con la siguiente iteración. Salta en la iteración *n*.

10. ¿Qué tipo de variables existen? ¿Es shell script fuertemente tipado? ¿Se pueden definir arreglos? ¿Cómo?

- Variables:
globales → por defecto
locales → se definen en funciones
de entorno → heredadas por los procesos hijos
- Declaración de arreglos:
arregloVacio=()
arregloLleno=(1 2 3 4 5) siempre sin espacios

11. Pueden definirse funciones dentro de un script? ¿Cómo? ¿Cómo se maneja el pasaje de parámetros de una función a la otra?

```
nombre_de_la_funcion() {  
    # Código de la función  
}
```

```
saludar_con_nombre() {  
    echo "Hola, $1, ¿cómo estás?"  
}
```

```
# Llamada a la función con un argumento  
saludar_con_nombre "Juan"
```

12. Evaluación de expresiones:

- Realizar un script que le solicite al usuario 2 números, los lea de la entrada Standard e imprima la multiplicación, suma, resta y cual es el mayor de los números leídos.
- Modificar el script creado en el inciso anterior para que los números sean recibidos como parámetros. El script debe controlar que los dos parámetros sean enviados
-
- Realizar una calculadora que ejecute las 4 operaciones básicas: +, -, *, %. Esta calculadora debe funcionar recibiendo la operación y los números como parámetros

13c.

```
if [ "$#" -ne 1 ]; then  
    echo "Uso: $0 <nombre_archivo>"  
    exit 1  
fi
```

Si el número de argumentos es distinto de 1 → se muestra un mensaje de uso que indica cómo se debe usar el script. `$0` representa el nombre del script en sí mismo.
`exit 1` → sale del script e imprime 1 indicando que hubo un error.

`if [-d "nombre"]` → pregunta si "nombre" es un directorio

`[-f "nombre"]` → devuelve V si existe el archivo, falso caso contrario

`mkdir` → crea carpetas o directorios

[\$? -eq 0] → verifica si el último comando se ejecuto correctamente. Si es así devuelve 0

for archivo in *; do:

el * → se refiere a todos los archivos y directorios en el directorio actual

15. COMANDO CUT

```
echo "Alice,25,New York" | cut -d ',' -f 1,3
```

Esto extraerá el primer y tercer campo, y la salida será "Alice,New York".

```
echo "Juan|45|Madrid" | cut -d '|' -f 1,3
```

Esto utilizará el carácter de tubería | como delimitador y extraerá el primer y tercer campo, resultando en "Juan|Madrid".

```
echo "Hola Mundo" | cut -c 1-4
```

Esto extraerá los caracteres del 1 al 4, resultando en "Hola".

```
echo "Juan|45|Madrid" | cut -d '|' --complement -f 2
```

Esto extraerá todos los campos excepto el segundo, resultando en "Juan|Madrid".