SCRIPTS DEL SHELL

MÓDULO 7

Sistemas Abiertos – Administración de SO I

1

Contenido del Módulo 7.

- Introducción.
  - Qué es un script, ventajas, interpretación y ejecución.
- Primeros pasos en bash scripting.
  - Estructura, comentarios, variables, operadores.
- Control de flujo en scripts.
  - Condicionales, bucles, funciones.
- Manejo de entrada y salida.
  - Lectura de entrada de usuario y archivos externos, parámetros.

2

Introducción.

- **¿Qué es un script?**
  - Conjunto de instrucciones en lenguaje interpretado, ejecutadas secuencialmente por un intérprete en lugar de un compilador.
  - En el contexto de SO GNU/Linux, el lenguaje es **Bash**, esto es, el intérprete de comandos tiene sus propia sintaxis y características.
  - Permiten automatizar tareas repetitivas y de administración del sistema. Al ser interpretados, no requieren ser compilados antes de ejecutarse.
  - Esto permite hacer cambios rápidos y ejecutar el código de inmediato, agilizando la prueba y ajuste del código.

3

Introducción.

- **¿Qué es un script?**
  - Los scripts pueden ir desde simples listas de comandos hasta programas complejos que incluyan **funciones avanzadas** y **estructuras de control de flujo**, como **bucles** y **condicionales**.
  - En un entorno Linux, escribir scripts puede ser esencial para tareas administrativas, monitoreo del sistema, manejo de archivos y mucho más.

```
#!/bin/bash
echo "Hola, mundo"
```

4

Introducción.

- **Ventajas de la automatización con scripts.**
  - La automatización de tareas mediante scripts ofrece múltiples beneficios:
    - **Eficiencia:** permiten realizar tareas repetitivas de forma rápida y sin intervención humana.
    - **Reducción de errores:** se minimiza el riesgo de errores humanos en tareas rutinarias, manteniendo la consistencia en los resultados.
    - **Flexibilidad:** son muy versátiles, pueden realizar tareas complejas,son fáciles de modificar y adaptar, sin necesidad de recompilar.

5

Introducción.

- **Ventajas de la automatización con scripts.**
  - La automatización de tareas mediante scripts ofrece múltiples beneficios:
    - **Mantenimiento y escalabilidad:** son fáciles de mantener y pueden evolucionar según los requerimientos del sistema. Pueden adaptarse a diferentes entornos o ampliarse.
    - **Automatización programada:** pueden programarse para ejecutarse en momentos específicos, sin necesidad de intervención humana constante. Esto permite realizar procesos de mantenimiento, monitoreo y respaldo de datos.

6

Introducción.

- **Interpretación y ejecución de scripts.**
  - La ejecución de scripts en GNU/Linux requiere entender cómo el sistema interpreta y procesa las instrucciones.
  - A continuación, se detallan los pasos fundamentales para escribir y ejecutar un script Bash en un entorno GNU/Linux:
    1. **Escribir el script:** Los scripts se escriben en un archivo de texto plano usando cualquier editor, como nano, vim o gedit.
       La primera línea del script debe ser el **shebang** (`#!/bin/bash`), que indica al sistema que el archivo será interpretado por Bash.

7

Introducción.

- **Interpretación y ejecución de scripts.**
    2. **Guardar el archivo y asignarle permisos de ejecución:** deben guardarse con un nombre descriptivo y, preferiblemente, con la extensión .sh para identificarlos fácilmente. Para permitir que el sistema ejecute el archivo como un programa, es necesario asignarle permisos de ejecución:

```
chmod +x mi_script.sh
```

8

Introducción.

- **Interpretación y ejecución de scripts.**
  3. **Ejecutar el script:** hay varias maneras:
     - Usando la ruta relativa del archivo.
     - Especificando la ruta absoluta.
     - Ejecutándolo con el intérprete directamente.

     Consejos para la ejecución de scripts:
     - Si se encuentra en un directorio declarado en PATH, se puede ejecutar simplemente usando su nombre.
     - Se puede modificar el archivo de configuración de shell, como `.bashrc` o `.profile`.

9

Introducción.

- **Interpretación y ejecución de scripts.**
  4. **Comprender el entorno de ejecución del script:**
     - Cada script se ejecuta en su propio entorno de Shell. Esto implica que las variables de un script no estarán disponibles fuera de él.
     - Los scripts pueden acceder a las variables de entorno del sistema, como $HOME, $PATH, y a los argumentos pasados al script mediante variables especiales como $1, $2, etc., que representan los parámetros en el orden en que se pasaron.

10

Introducción.

- **Interpretación y ejecución de scripts.**
    5. **Depurar y detener un script:**
        - Los scripts pueden detenerse con la combinación Ctrl+C en la terminal.
        - Para depurar, Bash ofrece opciones como -x para ver cada instrucción antes de ejecutarse.
        - Además, se pueden utilizar comandos de depuración como `echo` para imprimir el valor de variables en distintas etapas del script.

11

Primeros pasos en bash scripting.

- **Estructura de un script bash.**
    - Un script de bash está compuesto por los siguientes elementos:
        - **Shebang:** primera línea, indica al sistema qué interprete de comandos utilizar para ejecutar el script. Esta es `#!/bin/bash`.
        - **Comandos bash:** cualquier comando que puede ejecutarse en una terminal.
        - **Comentarios:** se indican con `#`, para documentar el script. No se ejecutan.
        - **Variables:** para almacenar valores que pueden cambiar durante la ejecución del script.

12

Primeros pasos en bash scripting.

- **Declaración de variables y tipos de datos básicos.**
  - En bash las variables se declaran y asignan sin especificar un tipo de dato. Por defecto, todo son cadenas de caracteres.
  - Para acceder a una variable se utiliza el `$` antes del nombre.
  - Las variables pueden ser:
    - Las de entorno, disponibles en todo el sistema.
    - Locales, disponibles solo dentro del script.
  - Aunque se trata todo como texto, se puede trabajar con números y operaciones matemáticas, con la notación `$((expresión))`.

13

Primeros pasos en bash scripting.

- **Declaración de variables y tipos de datos básicos.**

```
numero1=5
numero2=10
suma=$((numero1 + numero2))
echo "La suma es $suma"
```

14

Primeros pasos en bash scripting.

- **Operadores y expresiones aritméticas.**
  - **Operadores aritméticos:** Bash soporta los operadores básicos:

```
a=10
b=3
echo "Suma: $((a + b))"
echo "Resta: $((a - b))"
echo "Multiplicación: $((a * b))"
echo "División: $((a / b))"
echo "Módulo: $((a % b))"
```

15

Primeros pasos en bash scripting.

- **Operadores y expresiones aritméticas.**
  - **Operadores de comparación:** utilizados en estructuras de control de flujo de ejecución:
    - **-eq:** igual a
    - **-ne:** no igual a
    - **-gt:** mayor que
    - **-lt:** menor que
    - **-ge:** mayor o igual a
    - **-le:** menor o igual a.

```
a=5
b=10
if [ $a -lt $b ]; then
    echo "$a es menor que $b"
fi
```

16

Primeros pasos en bash scripting.

- **Operadores y expresiones aritméticas.**
  - **Operadores de comparación de cadenas:** útiles en scripts que manejan nombres de archivos o texto en general:
    - **=:** igual a
    - **!=:** no igual a
    - **<:** menor (alfabéticamente)
    - **>:** mayor (alfabéticamente)
    - **-z:** verdadero si la longitud de la cadena es cero.
    - **-n:** verdadero si la longitud de la cadena es distinta de cero.

17

Primeros pasos en bash scripting.

- **Operadores y expresiones aritméticas.**
  - **Operadores de comparación de cadenas:**

```
palabra="Hola"
if [ "$palabra" = "Hola" ]; then
    echo "La palabra es 'Hola'"
fi
```

18

Primeros pasos en bash scripting.

- **Buenas prácticas en la declaración de variables.**
  - Es importante que en un script:
    - **Nombres descriptivos:** el nombre del archivo describe su propósito.
    - **Evitar sobrescribir variables de entorno:** al declarar variables, no utilizar nombres que coincidan con las variables de entorno.
    - **Uso de mayúsculas en variables de entorno:** por convención, se utilizan mayúsculas para variables de entorno, y minúsculas para variables locales del script.
    - **Cerrar variables entre comillas:** Cuando pueden contener espacios o caracteres especiales, es importante encerrarlas entre comillas ("$variable") para evitar errores de interpretación.

19

Control de flujo en scripts.

- El control de flujo en Bash permite a los scripts tomar decisiones y ejecutar bloques de código en función de condiciones específicas.
- Los elementos básicos para el control de flujo en Bash son las estructuras **condicionales** y los **bucles**.
- Además, las **funciones** permiten estructurar el código de manera modular, facilitando su reutilización y mantenimiento.

20

Control de flujo en scripts.

- **Condicionales.**
  - Hay dos: **if** y **case**.
    - **Estructura if-else-elif:** estructura de selección básica, con la posibilidad de agregar condiciones con **elif**.

21

Control de flujo en scripts.

-

```
#!/bin/bash
numero=10

if [ $numero -gt 5 ]; then
    echo "El número es mayor que 5"
elif [ $numero -eq 5 ]; then
    echo "El número es igual a 5"
else
    echo "El número es menor que 5"
fi
```

22

Control de flujo en scripts.

- **Condicionales.**
  - **Estructura case:** útil cuando se quiere evaluar una variable en función de múltiples valores posibles.

23

Control de flujo en scripts.

- **Condicionales.**
  - **Estructura c** ar una variable en función de n

```bash
#!/bin/bash
opcion="b"

case $opcion in
    "a")
        echo "Opción A seleccionada"
        ;;
    "b")
        echo "Opción B seleccionada"
        ;;
    "c")
        echo "Opción C seleccionada"
        ;;
    *)
        echo "Opción no válida"
        ;;
esac
```

24

Control de flujo en scripts.

- **Bucles.**
  - Hay tres: **for** , **while** y **until**.
    - **Bucle for:** itera sobre una lista de elementos.

```
#!/bin/bash
for nombre in Juan Ana Pedro; do
    echo "Hola, $nombre"
done
```

25

Control de flujo en scripts.

- **Bucles.**
  - **Bucle while:** ejecuta un bloque de código mientras una condición se cumple.

```
#!/bin/bash
contador=1

while [ $contador -le 5 ]; do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

26

Control de flujo en scripts.

- **Bucles.**
  - **Bucle until:** similar a while, ejecuta un bloque de código pero mientras la condición **NO** se cumple.

```
#!/bin/bash
contador=1

until [ $contador -gt 5 ]; do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

27

Control de flujo en scripts.

- **Funciones en bash.**
  - Permiten estructurar el código de forma modular, encapsulando tareas específicas en bloques que pueden reutilizarse.
  - Esto facilita la lectura y el mantenimiento del script.
  - Se definen con la palabra clave `function` seguida del nombre y `{}`.

```
function nombre_funcion {
    # código de la función
}
```

28

Control de flujo en scripts.

- **Funciones en bash.**
  - Llamada a una función:

```
#!/bin/bash
function saludo {
    echo "Hola, bienvenido al script"
}

# Llamada a la función
saludo
```

29

Control de flujo en scripts.

- **Funciones en bash.**
  - Las funciones pueden aceptar parámetros, se acceden mediante `$1`, `$2`, etc., dentro de la función.
  - No es necesarios especificarlos en la definición, ya que bash los maneja de forma **posicional**.
  - Esto significa que, al llamar a la función, se pueden proporcionar diferentes cantidades de argumentos y la función simplemente asignará cada argumento en el orden en que se pasen, según la posición ($1, $2, etc.).

30

Control de flujo en scripts.

- **Funciones en bash.**
  - Llamada a una función con parámetros:

31

Control de flujo en scripts.

- **Funciones e**
  - Llamada

```bash
#!/bin/bash
function mostrar_parametros {
    echo "Parámetro 1: $1"
    echo "Parámetro 2: $2"
    echo "Parámetro 3: $3"
}

# Llamada con tres parámetros
mostrar_parametros "Hola" "Mundo" "!"

# Llamada con un solo parámetro
mostrar_parametros "Solo uno"
```

32

Control de flujo en scripts.

- **Funciones en bash.**
  - Bash no permite retorno explícito, se pueden utilizar las variables de entorno para los resultados, o imprimir el resultado en pantalla y capturarlo mediante sustitución de comandos `($(comando))`.

```
#!/bin/bash
function multiplicar {
    echo $(( $1 * $2 ))
}

resultado=$(multiplicar 5 4)
echo "El resultado de la multiplicación es $resultado"
```

33

Manejo de entrada y salida.

- En el módulo 3 vimos cómo se podía redirigir la entrada y la salida de los comandos. Esto también es válido en los scripts.
- En esta sección no focalizaremos en la lectura de entrada de usuario y archivos externos dentro de un script.
- **Lectura de entrada de usuario.**
  - Para capturar entradas del usuario en un script, utilizamos el comando `read`.
  - Este detiene la ejecución del script hasta que el usuario proporcione una entrada por teclado.
  - El valor ingresado se almacena en la variable especificada.

34

Manejo de entrada y salida.

- **Lectura de entrada de usuario.**

```
#!/bin/bash
echo "Ingrese su nombre:"
read nombre
echo "Hola, $nombre. Bienvenido al script."
```

35

Manejo de entrada y salida.

- **Lectura de entrada de usuario.**
  - Es posible capturar varios valores en una sola línea de entrada separando cada valor por un espacio.

```
#!/bin/bash
echo "Ingrese su nombre y apellido:"
read nombre apellido
echo "Nombre: $nombre"
echo "Apellido: $apellido"
```

36

Manejo de entrada y salida.

- **Lectura de entrada de usuario.**
  - La opción `-p` permite incluir un mensaje dentro del mismo comando `read`, lo que hace que el código sea más compacto.

```
read -p "Ingrese su edad: " edad
echo "Tienes $edad años."
```

37

Manejo de entrada y salida.

- **Lectura de entrada de usuario.**
  - Con la opción `-s` se oculta el texto ingresado por teclado, útil para el ingreso manual de contraseñas.

```
read -sp "Ingrese su contraseña: " password
echo -e "\nContraseña recibida."
```

38

Manejo de entrada y salida.

- **Lectura de archivos externos.**
  - Además de interactuar con el usuario, los scripts en Bash pueden leer datos directamente de archivos externos.
  - Una técnica común es usar un bucle `while` junto con `read`. Este enfoque permite procesar cada línea de forma secuencial.

```
#!/bin/bash
while read linea; do
    echo "Procesando: $linea"
done < archivo.txt
```

39

Manejo de entrada y salida.

- **Lectura de archivos externos.**
  - Podemos descomponer cada línea en varios campos, útil para archivos donde la línea tiene datos separados por espacios o comas.

```
#!/bin/bash
while IFS=',' read nombre edad ciudad; do
    echo "Nombre: $nombre, Edad: $edad, Ciudad: $ciudad"
done < datos.csv
```

40

Manejo de entrada y salida.

- **Lectura de archivos externos.**
  - Si el archivo que tenemos que procesar es pequeño y sabemos que contamos con suficiente memoria RAM para almacenarlo, resulta útil leerlo en una sola variable.

```
contenido=$(< archivo.txt)
echo "$contenido"
```

41

Manejo de entrada y salida.

- **Lectura de archivos externos.**
  - **Condicionales para control de archivos.**
    - La validación de archivos es crítica en un script de Bash.
    - Se dispone de los siguientes **operadores condicionales**:
      - **-e:** chequea la existencia del archivo.
      - **-f:** verifica si es un archivo regular (no un directorio, por ejemplo).
      - **-d:** comprueba si es un directorio.
      - **-x:** verifica si el archivo es ejecutable.
      - **-s:** chequea si el archivo no está vacío.
      - **-nt** y **–ot**: compara las fechas de modificación de dos archivos.

42

Manejo de entrada y salida.

- **Lectura de archivos externos.**
  - **Condicionales para control de archivos.**
    - Estos (y todos los) operadores se pueden utilizar en combinación con los operadores lógicos (`||`, `&&`) para realizar controles más complejos.

43

Manejo de entrada y salida.

- **Lec**
  -

```bash
#!/bin/bash

archivo="datos.txt"

# Verificar si el archivo existe y no está vacío
if [ -e "$archivo" ] && [ -s "$archivo" ]; then
    echo "El archivo '$archivo' existe y no está vacío. Procesando..."
    # Leer el archivo línea por línea
    while IFS= read -r linea; do
        echo "Línea: $linea"
    done < "$archivo"
else
    echo "El archivo '$archivo' no existe o está vacío."
fi
```

ación

más

44

Manejo de entrada y salida.

- **Parámetros externos al script.**
  - Un script de Bash puede recibir parámetros externos cuando se ejecuta desde la terminal.
  - Se utilizan **variables posicionales**, tal como en las funciones
  - Un script puede procesar varios parámetros, accediendo a ellos mediante sus posiciones:
    - **$1, $2, $3, ..., $n**: Los valores, según su posición.
    - **$@**: Lista de todos los parámetros, separados por espacios.
    - **$#**: Cantidad de parámetros pasados al script.

45

Manejo de entrada y salida.

- **Parámetros externos al script.**
  - Un script de Bash puede recibir parámetros externos cuando se

```
#!/bin/bash
echo "Primer parámetro: $1"
echo "Segundo parámetro: $2"
echo "Todos los parámetros: $@"
echo "Cantidad de parámetros: $#"
```

46

Manejo de entrada y salida.

- **Parámetros externos al script.**
  - También es posible pasar al script un archivo como parámetro.
  - A fin de cuentas, el parámetro siempre será una cadena de caracteres, en este caso la ruta al archivo.

47

Manejo de entrada y salida.

- **Parámetros e**
  - También e arámetro.
  - A fin de cu na de caracteres

```bash
#!/bin/bash

# Validar que se pase un parámetro
if [ $# -eq 0 ]; then
    echo "Uso: $0 archivo"
    exit 1
fi

# Asignar el parámetro a la variable archivo
archivo=$1

# Verificar si el archivo existe y no está vacío
if [ -e "$archivo" ] && [ -s "$archivo" ]; then
    echo "El archivo '$archivo' existe y no está vacío. Procesando..."
    # Leer el archivo línea por línea
    while IFS= read -r linea; do
        echo "Línea: $linea"
    done < "$archivo"
else
    echo "El archivo '$archivo' no existe o está vacío."
fi
```

48

Fin del Módulo 7

49