Sistemas Operativos II

Programación Shell-Script

- **Script** o programa shell
  - Fichero de texto conteniendo comandos externos e internos
    - **Secuencia de comandos** que se ejecutan en linea
    - **Variables y Entrada / Salida**
    - **Operadores** aritméticos y lógicos
    - Control de flujo
      - `if then, case in, …`
      - `for, while, until`
    - Funciones
    - Comentarios
  - Lo que aprendas del uso de **comandos desde la interfaz** te servirá en la programación de shell-scripts

- **Script** o programa shell
  - **Fichero** de texto conteniendo comandos externos e internos

```
$> cat welcome.sh
```

```
#!/bin/bash
echo "$USER, welcome to the real world"
exit 0
```

  - **#!** (*shebang)* indica el **intérprete de comandos** a usar por el script
    - El fichero **/etc/shells** contiene una lista con la ruta completa a los shells disponibles
  - Tiene la extensión **.sh** para indicar que es un archivo de script de shell

- **Script** o programa shell
  - **Ejecutar** un script desde la línea de comandos

```
$> ./welcome.sh
```

  - Requiere **permisos** de ejecución

```
$> chmod u+x welcome.sh
$> ls -l welcome.sh
rwxr--r-- 1 pablo citius 53 ene 29 18:24 welcome.sh
```

  - **Indicar** el **shell** como argumento
    - Sólo necesita permisos de lectura

```
$> bash welcome.sh
```

- Paso de **parámetros** al script
  - Podemos pasar argumentos desde la línea de comandos

```
$> ./welcome.sh [PARAMETROS]
```

  - Los **parámetros** se almacenan por orden en las **variables $n**
    - `$0` **nombre** del script
    - `$1 a $9` **parámetros** del 1 al 9
    - `${10}, ${11}, …` **parámetros** por encima de 9
    - `$#` **número** de parámetros
    - `$*, $@` **todos** los parámetros

- Paso de **parámetros** al script
  - `read x, echo $x`
    - Recuerda **convenio** mayúsculas para variables de entorno y minúsculas para variables locales
- Puedes usar variables del **entorno directamente** dentro del script
  - `echo $USER`
- Otras variables disponibles
  - **$?** código de salida del comando
  - **$$** indica el `PID` del script actual

- Estructura `if...then...fi`

```
if comando1
then
    ejecuta otros comandos
elif comando2
then
    ejecuta otros comandos
else
    ejecuta otros comandos
fi
```

- Estructura `if...then...fi`

  `if` `comando1`

  - Sólo chequea la salida de un comando (`$?`)
- Para **comparar valores** debemos usar el comando `test`
  - `test` *`expresión`* `o bien [` *`expresión`* `]`
    - `test` devuelve un **`código de salida 0`** si es correcto y un **`código de salida 1`** si es falso
    - `[` *`expresión`* `]` lleva espacios entre los corchetes

- Comando **`test`**
  - Algunas expresiones del comando `test`
    - `-d (comprueba que el parámetro es un directorio)`
    - `-f (comprueba que el parámetro es un fichero regular)`
    - `n1 -eq n2, n1 -gt n2, … (compara número enteros)`
    - `S1 = S2, S1 != S2 (compara/chequea string)`
  - Existen **expresiones** para chequear strings, números o ficheros
    - Ver material de apoyo al Tema 5: Uso del shell, sección 5.3
    - `$> man test`

- Comando **`test`**

```
num=4
if test $num -gt 5
then
  echo "$num if greater than 5"
else
  echo "$num if not greater than 5"
fi
```

  - También es válido
    - `if [ $num -gt 5 ]`
    - Espacio en blanco después de `[` y antes de `]` **obligatorios**

- Comando **`test`**
  - Operadores lógicos con test
  - **`!`** invierte el resultado de una expresión

```
if ! test $num -gt 5
if [ ! $num -gt 5 ]
```

  - **`-a, -o`** operador `AND`, `OR` respectivamente

```
if test $? -ne 0 -a $USER = 'root'
if [ $? -ne 0 -a $USER = 'root' ]
```

  - **`( expr )`** agrupación de expresiones

```
if test $? -ne 0 -a \( $USER = 'root' -o $USER = 'admin' \)
if [ $? -ne 0 -a \( $USER = 'root' -o $USER = 'admin' \) ]
```

    - Hay que eliminar el significado especial de los paréntesis
    - Espacios en blanco obligatorios

- Comando **`test extendido`**
  - A partir de la **versión 2.02** de Bash se introduce `[[ expr ]]`
  - Permite realizar **comparaciones** de un modo similar al de lenguajes estándar
    - Permite usar los **operadores** && y || para unir expresiones

```
if [ $? -gt 0 -a $USER = 'root' ]

if [[ $? -gt 0 && $USER = 'root' ]]
```

    - No necesita escapar los paréntesis

```
if [ $? -gt 0 -a \( $USER = 'root' -o $USER = ‘admin’ \) ]

if [[ $? -gt 0 && ($USER = 'root' || $USER = ‘admin’) ]]
```

- Estructura `case...in`

```
case valor in
    patrón1)
        bloque de comandos
        ;;
    patrón2)
        bloque de comandos
        ;;
    *)
        bloque de comandos por defecto
        ;;
esac
```

- Estructura `case...in`
  - ```
    echo -n "¿Quieres crear un directorio de test? (S/N):
    read respuesta
    case $respuesta in
         S | s)
              mkdir /tmp/test
              if [ $? -eq 0 ]
              then
                   echo 'Directorio creado correctamente'
              fi
              ;;
         N | n)
              echo 'No se ha creado ningún directorio'
              ;;
         *)
              echo 'Opción incorrecta'
              ;;
    esac
    ```
    ```

- Estructura `for`

```
for variable in lista
do
    bloque de comandos usando $variable
done
```

  - `variable` toma los valores de la lista en cada iteración
  - Podemos **usar** y **expandir variables** para crear la lista
  - `lista='1 2 3 4'`
    - `{1..4}`, `{a..d}`, `*.log`, `...`

- Estructura `for`

```
for fichero_log in /tmp/*.log
do
    rm $fichero_log
done


for sufijo in {00..05}
do
    touch servidor_${sufijo}.data
done
```

- Estructura `for`
  - Sintaxis **alternativa** similar a la del lenguaje C

```
for ((a=0; a < 5; a++))
do
    sufijo="0$a"
    touch servidor_${sufijo}.data
done
```

  - `$> touch servidor_0{0..5}.data`
    - Equivalente al bucle `for`

- Estructuras `while`

```
while comando
do
    bloque de comandos
done
```

```
while [ "$res" != "s" ]
do
    echo "¿Salir?"
    read res
done
```

- Estructuras until

```
until comando
do
    bloque de comandos
done
```

```
until [ "$res" = "s" ]
do
    echo "¿Salir?"
    read res
done
```

¿Por qué hay que poner $res entre comillas?

- Estructuras `for, while, until`
  - `break, continue`
    - Permite **salir** de un lazo **(`break`)** o **saltar** a la siguiente interación **(`continue`)**
    - `break n` puede salir de varios lazos a la vez

- Redirecciones
  - Podemos **guardar** la salida del script en un fichero
    - `echo $resultado > salida.data`
  - Y **leer** contenido de un fichero en el script
    - `read linea < error.log`
    - Sólo se lee la **primera linea** del fichero `error.log`
- Se pueden usar estas estructuras `for`, `while`, `until`, para leer el contenido de un fichero (**varias lineas**)

```
while read BUFFER
do
   echo "$BUFFER" >> $2
done < $1
```

- Funciones
  - Podemos **organizar el código** de un script con funciones

```
nombre_de_la_funcion() {
    comandos
}
```

  - Debemos definir la función **ANTES** de usarla en el script
  - Los **parámetros** se almacenan por orden en las **variables $n**
    - `$1 a $9` **parámetros** del 1 al 9
    - `${10}, ${11}, …` **parámetros** por encima de 9
    - `$*, $@` **todos** los parámetros
  - `${FUNCNAME[0]}` contiene el nombre de la función
    - `$0` sigue siendo el nombre del script

- Funciones
  - El **código de salida** se especifica con return

```
Nombre_de_la_funcion() {
    comandos
    return 0
}
```

  - **Por defecto** se devuelve el código del **último comando**
    - Es importante controlar el resultado de una función
  - Después de llamar a una función, **$?** tiene el código de salida

- Comentarios
  - Comentarios de **una** línea

```
# Script de ejemplo para aprender a programar
```

  - Comentarios de **varias** líneas
    - Preferible usar # en cada linea

```
# Script de ejemplo para aprender a programar
# Fecha: 21/01/2019
```

  - Comentarios de **varias** líneas

```
: '
Comentario de varias lineas
No es recomendable usarlo
porque puede introducir brechas de seguridad
'
```

  - Usar HereDoc es un truco usando redirecciones, no una forma real de escribir comentarios Bash multilínea (**no recomendado**)

- **Sustitución** de un **comando**
  - Permite usar la salida de un comando y **asignarla a una variable**
  - Usar símbolo especial ` (comilla aguda) o $()

```
x=`date`
echo $x
Mon Jan 21 08:19:24 CET 2020

DIR=$(pwd)
echo $DIR
/home/pablo
```

- Algunas consideraciones sobre **rendimiento**
  - El shell no es especialmente eficiente a la hora de ejecutar trabajos pesados
    - Usar **expansión aritmética** `$(( expr ))` o `$[ expr ]` en lugar del comando `expr` para evaluar expresiones
  - El uso de **lazos** para leer ficheros es bastante **ineficiente**
    - Intenta usar **comandos** (`wc, cat, grep, sed, …`)
  - Usa **comandos internos** del shell cuando sea posible
    - Reduce el **número de procesos** creados al ejecutar el script
  - Evita **Useless use of any command** (UUC)
    - `grep root /etc/passwd` ↔ `cat /etc/passwd | grep root`
  - Disminuye las redirecciones a fichero `>, >>`