Práctica 2: **Shell Scripting**

Sistemas Operativos

1 Introducción

1.1 Unix y GNU/Linux

Algunos conceptos previos que debes conocer o estar familiarizado antes de comenzar esta práctica incluyen la definición de **sistema operativo**, así como la clasificación de éstos. Con ello, podemos introducirnos en los sistemas operativos GNU/Linux, sistemas operativos multitarea y multiusuario, y que nosotros usaremos, por ahora, en su versión centralizada.

1.2 Editores de texto

Existen multitud de editores de texto que se utilizan por consola. Algunos ejemplos son `vi` o `emacs`. Si bien no es imprescindible, **sí es muy recomendable** usar uno de estos editores con soltura.

En Internet hay multitud de tutoriales de estos u otros editores de texto. Un sencillo manual de `vi` se puede encontrar en: `http://www.tutorialesytrucos.com/tutoriales/tutorial-unix-linux/26-tutorial-bsico-del-editor-vi.html`

2 El intérprete de comandos

El *intérprete de comandos*, o simplemente la *shell*, es una pieza software implementada en la mayoría de los sistemas operativos, y su función es proporcionar una interfaz de usuario para interpretar las órdenes introducidas por éste y acceder de esta manera a los distintos servicios proporcionados por el sistema operativo.

En general, distinguimos entre dos tipos de shell: *interfaz por línea de comandos* (o CLI: command-line interface) e *interfaz de usuario gráfica* (o GUI: graphical user interface).

En los sistemas operativos Linux, se pueden utilizar distintas shells CLI, como por ejemplo *sh*, *ksh* o *ash*. Sin embargo, probablemente la más conocida y utilizada en la actualidad es ***bash*** (acrónimo de Bourne-Again Shell),

y es el intérprete de comandos por defecto en la mayoría de distribuciones de Linux, así como en OSX. En cuanto a GUIs, algunas conocidas son GNOME, KDE o LXDE.

Si bien una GUI puede ser útil para diferentes tareas, las CLIs (en concreto ***bash***) es la herramienta aconsejable para otras muchas tareas, desde controlar servidores remotos hasta realizar operaciones en nuestra propia máquina, con especial énfasis en aquellas operaciones nativas para interactuar con el sistema operativo. En esta práctica nos centraremos en el uso de ***bash***.

Ejemplo 1. Abre un intérprete de comandos. ¿Qué información aparece en pantalla?
Habitualmente, encontrarás la secuencia `user@hostname:dir`, seguida del carácter $ (y finalmente de un cursor parpadeando).
Por ejemplo, la secuencia `jesus@lab:~` nos indica que el usuario actual es `jesus`, la máquina para la que el bash interpreta los comando se llama `lab` y el directorio actual es el *home* del usuario `jesus`. La forma en la que se muestra esta secuencia se puede cambiar en el fichero `~/.bashrc` o `~/.profile`.
El símbolo $ nos indica que el usuario actual es un usuario regular. En caso de superusuario (usuario *root*), se muestra el símbolo #.

2.1 Algunas teclas interesantes

Para interactuar con el intérprete de comandos, hay una serie de teclas que son indispensables conocer. Otras, en cambio, no son imprescindibles pero nos harán la vida más fácil:

- La tecla *enter*: Confirma la ejecución del comando escrito en el terminal.
- Las teclas ↑ y ↓ : Se usan para navegar en el historial de comandos introducidos (éstos quedan guardados en el fichero `~/.bash_history`).
- La tecla *tab*: Autocompletado (según contexto).
- El carácter ';' : Concatena varios comandos en una sola línea.
- El carácter '&' : Ejecuta el comando en background.
- El carácter '*' : Equivalente a cero o más caracteres en el nombre de los archivos.

- El carácter '?' : Equivalente a exactamente un carácter en el nombre de los archivos.
- El conjunto "[lista]" : Equivalente a exactamente un carácter del conjunto *lista*, en el nombre de los archivos.
- La combinación "Ctrl + c" : Interrumpe/Mata el proceso que se está ejecutado (con la señal SIGINT).

Ejemplo 2. Veamos algunos ejemplos con usando las teclas anteriores (más adelante veremos otros ejemplos con ellas).

1. Ejecuta el comando `hostname`, que muestra el nombre de la máquina.
2. Ejecuta el comando `hostinfo`, que muestra información sobre la máquina.
3. Ejecuta el comando `who`, que muestra información de los usuario conectados.
4. Usa las teclas ↑ y ↓ para navegar por el historial.
5. Ejecuta el comando `hostname; hostinfo`. ¿Qué ocurre en este caso?
6. Escribe en el prompt "`hostna`" y pulsa la tecla *tab*. ¿Qué ocurre?
7. Ahora escribe "`host`" y pulsa la tecla *tab*. ¿Qué ocurre en este caso?
8. Ejecuta el comando `sleep 2`, que *duerme* el proceso durante 2 segundos. ¿Qué ocurre en el prompt?
9. Ahora ejecuta el comando `sleep 2 &`. ¿Qué ocurre en este caso?
10. Ejecuta el comando `sleep 100`. Interrumpe esta última orden para volver tener el control del prompt.

2.2 El comando *clear*

Este comando nos sirve para limpiar la pantalla del terminal.

Ejemplo 3. Escribe el comando:

```
for i in {1..10}; do echo $i; done
```

que imprimirá por pantalla los números del 1 al 10. Tras eso, haz una limpieza del terminal usando el comando `clear`.

3 El sistema de ayuda

El sistema de ayuda se implementa de una forma estándar, y provee información sobre cada comando. Para invocarlo:

```
man [options] command
```

donde `command` es el comando del cual se requiere la información.

Ejemplo 4. ¿Cuáles son las opciones del comando `man`?
Teclea `man man`. Usa las teclas ↑ y ↓ para desplazarte por el contenido, y usa la tecla '`q`' para salir y volver al *prompt*.

Otra forma de invocar la ayuda es usando palabras clave:

```
man -k key
```

Ejemplo 5. Imagina que quieres compilar un fichero `java` pero no sabes el nombre del compilador de Java. Usa el comando anterior para averiguarlo.

4 El sistema de ficheros

El sistema de ficheros son las estructuras lógicas así como los métodos para gestionarlas que utiliza el sistema operativo en la gestión y organización de los ficheros dentro de un soporte físico. Estos métodos que utiliza el sistema operativo para pasar del soporte físico (disco duro) al soporte lógico (particiones), y viceversa, lo hace de una forma transparente al usuario final (y además, no es parte del temario de esta asignatura . . .).

En el caso de los sistemas Linux (como en muchos otros), el sistema de ficheros está organizado jerárquicamente en forma de árbol. Esto significa que los archivos son agrupados en directorios, los cuales también pueden estar contenidos en otros directorios (siendo subdirectorios de éste), hasta un directorio raíz que es "/". A partir de él, se organizan los directorios más importantes:

- /bin : programas que estarán disponibles en modos de ejecución restringidos (bash, cat, ls, ps, . . .).
- /boot : kernel (núcleo del sistema) y ficheros de arranque.
- /dev : dispositivos externos.
- /etc : archivos de configuración.

- /home : usuarios del sistema.
- /lib : librerías indispensables y otros módulos.
- /proc : directorio virtual para el intercambio de ficheros.
- /root : administradores (superusuarios, o usuarios *root*) del sistema.
- /sbin : programas que estarán disponibles para usuarios *root* en modos de ejecución restringidos.
- /tmp : archivos temporales (el directorio es vaciado en el arranque).
- /usr : programas accesibles a todos los usuarios (y algunos ficheros no modificables que usan estos programas).
- /var : archivos que son modificados frecuentemente por otros programas.

Ejemplo 6. Muestra el contenido del directorio raíz ejecutando el comando:

```
ls /
```

¿Qué ocurre al mostrar el contenido del `/home`? Para ello, ejecuta el comando:

```
ls /home
```

La **ruta** es la cadena de texto única que identifica a cada fichero. Se construye concatenando los nombres de los directorios, en orden jerárquico, hasta llegar al directorio que contiene el fichero, y concatenando finalmente el nombre del fichero correspondiente.

4.1 Rutas relativas y absolutas

Las rutas se pueden especificar de forma relativa o de forma absoluta.

Cuando hablamos de **rutas absolutas**, nos referimos a la concatenación completa de todos los directorios y subdirectorios desde "/" hasta el fichero al que apunte la ruta correspondiente.

En el caso de **rutas relativas**, se hace únicamente referencia a la jerarquía de subdirectorios a partir del directorio actual.

Existen también algunos caracteres reservados:

- Carácter "." : referencia el directorio actual.

- String ".." : referencia al directorio donde está contenido el directorio actual.
- Carácter ~ : referencia al *home* del usuario.

Ejemplo 7. Hemos iniciado sesión con el usuario `esthercolero` y estamos en su *home* (`/home/esthercolero/`). Veamos los siguientes ejemplos:

- La ruta relativa `~/foo.txt`
  apunta a la ruta absoluta `/home/esthercolero/foo.txt`
- La ruta relativa `./foo2.txt`
  apunta a la ruta absoluta `/home/esthercolero/foo2.txt`
- La ruta relativa `./../esthercolero`
  apunta a la ruta absoluta `/home/esthercolero`
- ¿A qué ruta absoluta apunta la ruta relativa
  `~/.././esthercolero/./../dir1/../`?

5 Comandos para el manejo de ficheros

En esta sección veremos los comandos básicos para el manejo de ficheros. Aquí se explican en su forma básica; pero todos incluyen opciones adicionales que pueden ser consultadas accediendo a la ayuda del comando correspondiente.

5.1 El comando *pwd*

El comando `pwd` imprime el nombre del directorio de trabajo actual.

```
pwd
```

En general, por defecto la shell inicializa su directorio de trabajo actual al *home* del usuario.

5.2 El comando *cd*

El comando `cd` cambia el directorio de trabajo a la ruta especificada por argumentos:

```
cd <path>
```

Cuando se utiliza sin argumentos, cambia el directorio de trabajo al *home* del usuario. Es decir, el comando `cd` es equivalente al comando `cd ~`.

5.3 El comando *mkdir*

El comando `mkdir` crea el directorio especificado por argumentos:

```
mkdir <path>
```

5.4 Los comandos *cp* y *mv*

Los comandos `cp` y `mv` respectivamente copian o mueven el contenido de `<source>` a `<target>`:

```
cp <source> <target>
mv <source> <target>
```

5.5 Los comandos *rm* y *rmdir*

El comando `rm` elimina el fichero especificado por argumentos:

```
rm <path>
```

En el caso de que el fichero especificado en `<path>` sea un directorio, es necesario invocar al comando `rmdir`. Sin embargo, para que su eliminación sea satisfactoria, es necesario que el directorio esté vacío.

```
rmdir <path>
```

5.6 El comando *ls*

El comando `ls` muestra el contenido de la ruta especificada por argumentos:

```
ls <path>
```

En caso de no introducir ningún argumento, este comando muestra el contenido del directorio actual. Es decir, el comando `ls .` es equivalente al comando `ls`

Ejemplo 8. Razona el comportamiento de ejecutar los siguientes comandos (en dicho orden):

1. `cd`
2. `cp foo.txt myfile.pdf`
3. `mv foo2.txt otherfile.jpg`
4. `mkdir ejemplo`

5. `cd ejemplo`
6. `cp ../myfile.pdf .`
7. `cd ..  ; mv otherfile.jpg ./foo2.txt`
8. `rmdir ejemplo`
9. `rm ejemplo/myfile.pdf ; rmdir ejemplo`

6 Variables, Condiciones y Bucles

Cuando se está usando la shell, a menudo es necesario hacer uso de variables, condiciones y bucles, los cuales nos permiten ejecutar comandos de una forma más apropiada para nuestros propósitos.

En el caso de las variables, distinguiremos entre variables locales (también conocidas como *User Defined Variables*), que son creadas y gestionadas por el usuario, y las variables de entorno (o *System Variables*), que son creadas y gestionadas por el sistema operativo.

6.1 Variables locales

En general, estas variables comienzan por letras minúsculas (para diferenciarlas de las variables de entorno). La forma de definirlas es la siguiente:

```
variableName=value
```

donde `value` es un *String*.

Algunos aspectos interesantes a considerar de la definición anterior:

- El nombre de la variable `variableName` sólo acepta caracteres alfanuméricos y la barra baja '_'.
- Los nombre de las variables son sensibles a mayúsculas/minúsuclas (*case-sensitive*).
- Los espacios no son admitidos en la definición de una variable.
- Una variable puede recibir el valor `NULL` definiéndola como:
  `var=` o `var=""`

Para usar su valor en la invocación a otros comandos, hay que añadir el carácter '$' antes del nombre de la variable.

Ejemplo 9. Razona el comportamiento de ejecutar los siguientes comandos (en dicho orden):

1. `a=1`
2. `A=6`
3. `b = 30`
4. `echo $a`
5. `echo $A`
6. `sleep $a ; echo "hello!" ; sleep $A ; echo "bye!"`

Dado que el `value` asignado a una variable es un *String*, para ciertas operaciones aritméticas es necesario el uso de otros comandos como `define`, `let` o `expr`.

6.2 Variables de entorno

Estas variables son proporcionadas por el sistema, y su utilidad puede ser distinta dependiendo de cada una de ellas. Por ejemplo, algunas son útiles para no tener que escribir mucho al utilizar un programa; otras son usadas por la propio *shell*, etc...

Aquí listamos **algunas** de las más importantes:

- HOME : Muestra la ruta al *home* del usuario.
- USER : Muestra el nombre del usuario.
- PATH : Muestra el contenido del *path* del usuario.
- SHELL : Muestra la *shell* por defecto del sistema.
- ...

Al igual que las variables locales, las variables de entorno pueden cambiar su valor (comandos `set`, `export`). Sin embargo, asignar un valor no adecuado a algunas de estas variables puede crear problemas en el sistema.

Ejemplo 10. Razona el comportamiento de ejecutar los siguientes comandos:

1. `echo $HOME`
2. `echo $PATH`

6.3 Condiciones

Las condiciones tienen la siguiente sintaxis:

```
if condition
then
       commands....
else
       commands....
fi
```

donde la condición se evalúa a `true` cuando una expresión es cierta o cuando recibe 0 como *exit status* de un comando.

En caso de **expresiones**, deben **escribirse entre corchetes y con un espacio**. Distinguimos entre expresiones aritméticas, expresiones sobre cadenas de texto, expresiones sobre ficheros y expresiones lógicas.

Cuando se usan los condicionales en una sola línea, se debe añadir el carácter ';' después de la condición.

6.3.1 Expresiones aritméticas

Para evaluarlas se usan los operadores `x -eq y` (x==y), `-ne` (!=), `-lt` (<), `-le` (<=), `-gt` (>) y `-ge` (>=).

6.3.2 Expresiones sobre strings

Para evaluarlas se usan los operadores `s1 = s2` (mismo valor), `s1 != s2` (distinto valor), `s1` (no definido o no NULL), `-n s1` (existe y no NULL) y `-z s1` (existe y NULL).

6.3.3 Expresiones sobre ficheros

Para evaluarlas se usan los operadores `-s f` (fichero `f` no vacío), `-f f` (`f` existe), `-d d` (`d` es directorio), `-w f` (`f` tiene permisos de escritura), `-r f` (`f` tiene permisos de lectura), y `-x f` (`f` tiene permisos de ejecución).

6.3.4 Expresiones lógicas

En las expresiones, se pueden usar los operadores '!' (NOT), '-a' (AND) y '-o' (OR).

También es posible usar los operadores "&&" (AND) y "||" (OR) para separar expresiones.

Ejemplo 11. Veamos el siguiente ejemplo donde usamos todas las expresiones anteriores:

```
a=5;
if [ $a -le 6 -a $USER == estercolero -a -f foo.txt ]
       && [ -d ejemplo ];
then
       echo "hello!"
fi
```

6.3.5 Exit status

El *exit status* de un comando es el valor que éste devuelve a la *shell*, y que nos indica si la ejecución de dicho comando fue satisfactoria o no. En general, una ejecución satisfactoria devuelve el valor 0.

Ejemplo 12. Por ejemplo, el comando `cat foo` devolverá un *exit status* con valor 0 si el fichero `foo` existe; en caso contrario devolverá > 0.
¿Qué devuelve el siguiente comando?
`if cat foo || !  cat foo; then echo "yes"; else echo "no"; fi`

Además, en la variable `$?` queda almacenado el *exit status* del último comando ejecutado.

Ejemplo 13. Crear el siguiente fichero `exit.c`:

```
#include <string.h>
int main(int argc, char* argv[]){
        if(argc >= 2){
                if(!strcmp(argv[1], "ten")){
                        return 10;
                }else if (!strcmp(argv[1], "twenty")){
                        return 20;
                }
        }
        return 100;
}
```

y compílalo para generar el ejecutable `exit`:

```
gcc exit.c -o exit
```

Ahora razona el comportamiento de los siguientes comandos:

- `if ./exit ten == 10; then echo "ten";`
  `else echo "unknown"; fi`
- `./exit twenty; e=$?;`
  `if [ $e == 10 ]; then echo "ten";`
  `elif [ $e == 20 ]; then echo "twenty";`
  `else echo "unknown"; fi`

¿Qué ocurre si ejecutamos lo siguiente?

```
./exit twenty;
if [ $?  == 10 ]; then echo "ten";
elif [ $?  == 20 ];then echo "twenty";
else echo "unknown"; fi
```

6.4 Bucles

6.4.1 Bucle FOR y listas

Tienen la siguiente sintaxis:

```
for <varName> in <list>
do
          commands....
done
```

Ejemplo 14. En el siguiente ejemplo se muestran dos bucles anidados:

```
for n in 1 2 3 4 5 6 7 8 9 10
do
    for i in 1 2 3 4 5 6 7 8 9 10
    do
         echo "$n * $i = $(expr $i \* $n)"
    done
done
```

6.4.2 Bucle FOR y expresiones

Tienen la siguiente sintaxis:

```
for (( expr1; expr2; expr3 ))
do
          commands....
done
```

de forma que `expr1` es evaluada en la primera iteración (usualmente para inicializar las variables del bucle), `expr2` es evaluada en todas las demás (condición de fin), y `expr3` se evalúa al final de cada iteración (usualmente para incrementar el valor de las variables del bucle).

Ejemplo 15. En el siguiente ejemplo se muestran dos bucles anidados:

```
for (( i = 1; i <= 5; i++ ))
do
    for (( j = 1 ; j <= 5; j++ ))
    do
         echo -n "$i "
    done
    echo ""
done
```

6.4.3 Bucle WHILE y expresión

Tienen la siguiente sintaxis:

```
while [ condition ]
do
          commands....
done
```

donde `condition` tiene la misma sintaxis que para los *if-else-fi*.

7 Comandos para el manejo de la entrada/salida

7.1 El comando *echo*

El comando `echo` escribe el contenido de los argumentos por la salida estándar:

```
echo <string>
```

Este comando también acepta variables.

Ejemplo 16. ¿Qué diferencia hay entre las siguientes líneas?

```
a=10 ; b=3
echo a es $a y b $b
echo 'a es $a y b $b'
echo "a es $a y b $b"
```

7.2 El comando *cat*

El comando `cat` escribe el contenido del fichero pasado por argumento (por la salida estándar):

```
cat <file>
```

7.3 Los comandos *head* y *tail*

Los comandos `head` y `tail` respectivamente muestran las primeras/últimas líneas de un fichero (por la salida estándar):

```
head <file>
tail <file>
```

8 Permisos, usuarios y grupos

Una parte muy importante del sistema operativo es la gestión de los permisos de los ficheros. En los sistemas GNU/Linux, se establecen 3 tipos de permisos (lectura, escritura y ejecución) en tres categorías (el propietario del fichero, grupo del propietario del fichero, y cualquier usuario).

Recalcar que los usuarios se organizan en grupos, de forma que un grupo puede contener múltiples usuarios. Además, el SO permite crear nuevos grupos y la gestión de los usuarios en los distintos grupos es sencilla para el administrador del sistema (pero eso no lo veremos en esta práctica ...).

Para ver cómo se organizan los permisos de un fichero, analicemos la siguiente secuencia de 9 caracteres:

```
r w x r - x r - -
```

Cada grupo de tres caracteres representan los permisos de: (i) el usuario propietario (es decir, los permisos `rwx`), (ii) de todos los usuarios del mismo grupo que el propietario (es decir, los permisos `r-x`), y (iii) de cualquier otro usuario (es decir, los permisos `r--`). Estas tres categorías de usuarios las declararemos con los caracteres `u` (*user*), `g` (*group*) y `o` (*others*). Además, el carácter `a` (*all*) nos permitirá referirnos a las tres categorías a la vez.

En cuanto a las ternas de tres caracteres, éstas representan los permisos de lectura `r` (*read*), escritura `w` (*write*) y ejecución `x` (*execute*), del grupo correspondiente. El carácter '-' representa la ausencia del permiso correspondiente.

Ejemplo 17. En el ejemplo anterior, el propietario tiene permisos de lectura, escritura y ejecución; el grupo tiene permisos de lectura y ejecución, y el resto de usuarios sólo tiene permisos de lectura.

Para obtener esta información basta ejecutar el comando `ls -l` (es decir, en formato largo). Cada fichero aparece en una fila. Para cada uno, el primer carácter representa el tipo de fichero (directorio, enlaces, ficheros regulares, ...), y los 9 siguientes representan los permisos. También aparece otra información como el propietario y el grupo, así como el tamaño del fichero.

8.1 El comando *chmod*

El comando `chmod` sirve para cambiar los permisos del fichero pasado por argumentos:

```
chmod <permissions> <file>
```

de forma que `<permissions>` tiene la siguiente sintaxis:

```
<permissions> := <sentence> [,<sentence>]*
<sentence>    := <group> ('+' | '-' | '=' (<type>)+)+
<group>       := 'u' | 'g' | 'o' | 'a'
<type>        := 'r' | 'w' | 'x'
```

Hay que remarcar que el anterior comando no modifica aquellos permisos a los que no se haga referencia.

Ejemplo 18. El siguiente comando asigna para el propietario permisos de lectura y ejecución y se los quita para la escritura, para el grupo asigna permisos de ejecución, y para los otros les deja únicamente permisos de lectura (es decir, en caso de tener previamente permisos de escritura y ejecución, se los quita):

```
chmod u+rx-w,g+x,o=r foo
```

Otra forma de usar el comando `chmod` es introduciendo los permisos en formato octal, de forma que para tripleta de permisos, se calcula un número como suma de:

```
r w x
4 2 1
```

De esta forma, los permisos de lectura+escritura+ejecución tienen un valor de 7, los permisos de lectura+ejecución tienen un valor de 5, los permisos de lectura+escritura tienen un valor de 6, etc...

En el comando `chmod` necesita como argumento un número de tres dígitos, que corresponden con los permisos del usuario, grupo y otros, sucesivamente.

Ejemplo 19. Similar al ejemplo anterior podemos ejecutar el siguiente comando:

```
chmod 514 foo
```

¿Cuál es la diferencia con el ejemplo anterior¿

9 Manipulación de strings y otros operadores

Otra opción que también nos permite *bash* es manipular fácilmente los strings (es decir, las variables). Para usarlos, encerraremos entre llaves { y }. Es decir, la variable `$var` la escribiremos como `${var}`. Veamos algunas de las manipulaciones que podemos conseguir:

La **longitud** de un string se puede extraer con el operador '#':

Ejemplo 20. Longitud de un string:

```
var=abcABC123ABCabc
echo ${#var}      #15
```

Se pueden extraer **substrings** con el operador ':' e indicando la posición inicial y la longitud:

Ejemplo 21. Substring:

```
var=abcABC123ABCabc
echo ${var:0}      #abcABC123ABCabc
echo ${var:6}      #123ABCabc
echo ${var:9:3}    #ABC
```

También se pueden remover **substrings** usando patrones:

- `${var#pattern}` : elimina de `var` el *string* más corto igual a `pattern` desde el principio.
- `${var##pattern}` : elimina de `var` el *string* más largo igual a `pattern` desde el principio.

- `${var%pattern}` : elimina de `var` el *string* más corto igual a `pattern` desde el final.
- `${var%%pattern}`: elimina de `var` el *string* más largo igual a `pattern` desde el final.

Ejemplo 22. Substring:

```
var=abcABC123ABCabc
echo ${var#a*C}      # 123ABCabc
echo ${va##a*C}      # abc
echo ${va%b*c}       # abcABC123ABCa
echo ${va%%b*c}      # a
```

Otra forma posibilidad muy interesante es guardar el **resultado de la ejecución de algún comando**. Por ejemplo, imagina que necesitamos guardar en alguna variable el directorio de trabajo actual. Esto nos los proporciona el comando `pwd`. Sin embargo, para guardarlo en una variable, necesitamos usarlo entre los operadores `$(...)`. Otra opción es usarlos entre los operadores `` `...` ``.

Ejemplo 23. Ejecución de comandos:

```
myDir=$(pwd)
```

10 Bash Script Files

Todo el contenido de esta práctica (comandos, condiciones, bucles, ...) puede usarse de forma conjunta y guardarse en un único fichero, cuyo nombre es *bash script*. La utilidad de estos ficheros es guardar una secuencia de operaciones que a menudo se realizará de forma conjunta, y de esta forma, ahorramos tener que escribirlos una y otra vez.

Habitualmente, estos ficheros tienen la extensión `sh`, y para ejecutarlos deben tener los permisos adecuados.

10.1 Argumentos en Scripts

En la *shell*, podemos referirnos a los argumentos pasados por parámetros usando el operador `$i`, donde $i \geq 0$. En concreto, el nombre del programa se encuentra en `$0`, y el resto de parámetros en `$1`, `$2`, ...

Ejemplo 24. En este ejemplo usaremos el fichero `test.sh`, cuyo contenido es el siguiente:

```
if [ $1 == suma ]
then
       echo "$2 + $3 = $(expr $2 + $3) .";
elif [ $1 == resta ]
then
       echo "$2 - $3 = $(expr $2 - $3) .";
else
       echo "No entiendo arg1 = $1 ."
fi
```

y que tiene los permisos necesarios para ser ejecutado.
¿Qué producen las siguientes llamadas?

1. `./test.sh suma 10 15`
2. `./test.sh resta 10 3`
3. `./test.sh divide 10 15`

11 Ejercicios

Ejercicio 0. Uso de la ayuda: Utiliza el sistema de ayuda para ver las opciones principales de los comandos de manejo de ficheros y manejo de entrada/salida vistos en esta práctica.

Ejercicio 1. Manejo de ficheros: Realiza las siguientes tareas usando un comando por tarea (en algunas hará falta uso de bucles, condicionales, etc...). Si es necesario, usa las opciones adicionales de los comandos estudiados en esta práctica (ver ejercicio anterior).

1. Crea el directorio `practica2`.
2. Entra en él.
3. Crea los directorio `dir_1`, . . . , `dir_5` dentro de `practica2`.
4. Crea los directorios `carpeta1`, ..., `carpeta10` dentro de `dir_1`.
5. Crea los ficheros `foo1.txt`, `foo2.txt`, `foo.c` y `README` usando un editor de texto (no importa su contenido).
6. Copia los ficheros con extension `txt` al directorio `dir_2`.
7. Copia los ficheros `foo1.txt`, `foo2.txt` y `foo.c` al directorio `dir_3`.

8. Mueve el fichero `foo.c` al directorio `dir_4` con el nombre `kk.c`.
9. Copia el directorio `dir_1` (y todo su contenido) en el directorio `copia`.
10. Elimina el fichero `README`.
11. Elimina el directorio `dir_5`
12. Elimina el directorio `dir_1`
13. Muestra todo el contenido del directorio de trabajo actual (`practica2`).
14. Muestra el contenido de todos los directorios contenidos en el directorio actual (usa bucles y condicionales).

Ejercicio 2. Manejo de entrada/salida: Realiza las siguientes tareas usando un comando por tarea (en algunas hará falta uso de bucles, condicionales, etc...). Si es necesario, usa las opciones adicionales de los comandos estudiados en esta práctica (ver ejercicio anterior).

1. Crea el directorio `practica2` (si no existe), y entra en él.
2. Imprime el siguiente mensaje:
   `Está en el directorio <dir> y su nombre de usuario es <user>.`
   donde `<dir>` y `<user>` deben ser correctamente completadas.
3. Ejecuta el siguiente comando:
   `for i in {1..100}; do echo $i >> numbers; done`
   que crea el fichero `numbers`, que contiene 100 líneas, cada una de las cuales tiene el número de línea del fichero.
4. Muestra todo el contenido del fichero.
5. Muestra la 3 primeras líneas del fichero.
6. Muestra las 15 últimas líneas del fichero.
7. Imprime el valor de cada línea multiplicado por 10.

Ejercicio 3. Permisos: Realiza las siguientes tareas usando un comando por tarea.

1. Crea el directorio `practica2` (si no existe), y entra en él.
2. Si no existe, crea el fichero `README` (sin importar su contenido).

3. Usa el comando `ls -l README` para ver los permisos actuales del fichero.

4. Añade permisos de lectura para el grupo en este fichero (sin modificar el resto). Usa el comando anterior para ver que se han modificado correctamente.

5. Añade permisos de escritura para el propietario y para *otros*, y elimina los permisos de ejecución para estos últimos (sin modificar el resto). Comprueba el cambio.

6. Añade permisos de ejecución para el propietario, y asigna únicamente permisos de lectura al grupo (sin modificar el resto). Comprueba el cambio.

7. Modifica los permisos de ese fichero para que el usuario pueda leer, escribir y ejecutar, el grupo sólo pueda leer y ejecutar, y los *otros* sólo puedan leer. Comprueba el cambio.

Ejercicio 4. Script Files: Crea el fichero `myscript.sh`, con permisos de ejecución para cualquier usuario, y que realice las siguientes operaciones:

1. Comprueba si existe el fichero pasado en el primer argumento, e imprimirá un mensaje acorde.

2. Crea el directorio pasado en el segundo argumento y entre en él.

3. Copia el fichero pasado en el primer argumento (asumiremos en este caso que sí existe) al directorio actual.

4. Crea tantos directorios con el nombre `test<i>` (donde `i` abarca desde el 1 hasta el número indicado en el tercer argumento), y dentro de cada uno de ellos, crea los subdirectorios `sub-<j>-test-<i>` (donde `j`, al igual que `i` viene determinado por el cuarto argumento).

5. Muestra el mensaje: `El argumento 5 es <arg5> y el argumento 6 es <arg6>`.

6. Vuelva al directorio inicial.

Entrega de la Práctica 2: Shell Scripting

Instrucciones de la entrega

La práctica consta de una serie de problemas, que valen el 70% de la nota de la práctica, y un test adicional que vale el 30%. La práctica se realizará **individualmente** y se entregará en un fichero PDF, que contenga las soluciones a los problemas aquí propuestos. La entrega se realizará adjuntando dicho fichero a la tarea correspondiente que se abrirá en el Campus Virtual. Las instrucciones para la realización del test adicional se describen a continuación.

Fecha de entrega

La entrega finalizará el día anterior a la primera sesión de la Práctica 3; es decir:

- Grupos A, B y F: Martes 29 de marzo.
- Grupo C: Miércoles 30 de marzo.

Test Adicional

Adicionalmente, el día siguiente a la fecha de entrega (es decir, en la primera sesión de la práctica 3) se realizará un cuestionario tipo test, que obligatoriamente será realizado en las aulas de prácticas y de forma individual. Este test cuenta el 30% de la nota de esta práctica.

La no realización del test o la realización del mismo fuera de las aulas conlleva que éste automáticamente califique como 0.

Plagio

En caso de copia parcial o completa de la entrega de esta práctica entre varios estudiantes, la práctica quedará calificada como 0, para todas las personas involucradas (es decir, tanto para el copiador como para el que se dejó copiar).

En caso de duda, los profesores podrán pedir una serie de ejercicios adicionales, que serán realizados en persona.

Problemas:

Problema 1. Manejo de ficheros. (**1.5 puntos**)

Realiza las siguientes tareas usando un comando por tarea (en algunas hará falta uso de bucles, condicionales, etc...). Si es necesario, usa las opciones adicionales de los comandos estudiados en esta práctica.

1. Crea el directorio `problema2`.
2. Entra en él.
3. Crea el directorio `test` siempre y cuando no exista en el directorio superior el fichero `foo`.
4. Crea los 4 directorios `test3`, ..., `test6`, y dentro de todos ellos los 7 directorios `subtest2`, ..., `subtest8`.
5. Imagina que en el directorio `test1` se encuentran los ficheros `main.c`, `sthg.c`, `sthg.h`, `sthg2.c` y `sthg2.h`. Copia todos los archivos con extensión `c` al directorio `test2`.
6. Cambia el nombre del fichero `main.c` del directorio `test2` por `newmain.cpp`.
7. Elimina todos los ficheros con extensión `h` del directorio `test1`.
8. Elimina el directorio test5 y todo su contenido.
9. Muestra el contenido de todos los directorios contenidos en el directorio actual.

Problema 2. Manejo de entrada/salida. (**1.5 puntos**)

Realiza las siguientes tareas usando un comando por tarea (en algunas hará falta uso de bucles, condicionales, etc...). Si es necesario, usa las opciones adicionales de los comandos estudiados en esta práctica.

1. Imprime la ruta absoluta del *home* del usuario.
2. Muestra el contenido del fichero `kk` enumerando sus líneas.
3. Muestra las 6 primeras líneas de los ficheros `kk1` y `kk2`.
4. Considera el fichero `integers`, que en cada línea contiene un número entero. Imprime el valor de cada línea dividido por 7.

Problema 3. Permisos. (**1 punto**)

Añade permisos de lectura para el propietario y para *otros*, y elimina los permisos de escritura para estos últimos, y establece para el grupo únicamente permisos de ejecución (sin modificar el resto) en el fichero `foo`. Usa un **único** comando para realizarlo.

Problema 4. Permisos. (**1 punto**)

Establece los siguientes permisos en el fichero `foo`: usuario con permisos de escritura y ejecución, grupo con permisos de lectura y ejecución, y otros con permiso de lectura y escritura. Usa un **único** comando para realizarlo.

Problema 5. Script Files. (**2 puntos**)

Escribe el contenido de un script que realice las siguientes operaciones:

1. Inicialmente comprobará que el fichero pasado en el primer argumento existe, en caso contrario, mostrará un error.

2. En caso de que exista, procede con las siguientes operaciones (en caso de que no exista, termina).

3. El contenido de ese fichero es una única línea con el siguiente formato:

   $$<\text{string1}>,<\text{string2}>,\ldots<\text{stringN}>,$$

   Asumiremos que todos los ficheros de entrada contienen un formato correcto.

4. Se pide imprimir una línea para cada uno de esos *strings* (es decir, `N` líneas), de forma que cada una de ellas contenga k repeticiones, separadas por un espacio, del *string* n-ésimo. Es decir:

   $$\begin{gathered}<\text{string1}>^1\ <\text{string1}>^2\ \ldots<\text{string1}>^k\\ <\text{string2}>^1\ <\text{string2}>^2\ \ldots<\text{string2}>^k\\ \ldots\\ <\text{stringN}>^1\ <\text{stringN}>^2\ \ldots<\text{stringN}>^k\end{gathered}$$

   El número k será especificado en el segundo argumento del script.