Práctica 1

Programación en lenguaje *Shell*

Objetivo

Familiarizarse con el lenguaje de programación *shell*, concretamente en su variante **Bash**.

1.1. Introducción

Una *shell* es un programa que actúa como interfaz entre el usuario y el sistema operativo. Permite a aquél introducir órdenes para que sean ejecutadas por éste.

Existen dos modos posibles de trabajar con una *shell*. La primera forma es mostrando un cursor o *prompt* que permite al usuario escribir directamente las órdenes, de tal manera que sean interpretadas y ejecutadas inmediatamente por el sistema operativo. Se implementa un bucle para poder ejecutar las órdenes de forma inmediata según los siguientes pasos:

```
Cursor -> Introducir comando -> Interpretar comando
-> Ejecutar comando -> Salida de comando -> Cursor -> ...
```

La segunda forma es lanzando los comandos a través de un guión o *script*. Básicamente, un guión es un fichero de texto cuyo contenido son órdenes del sistema operativo y estructuras de control similares a las usadas en lenguajes con programación estructurada como Pascal o C. Asignándole permisos de ejecución a este fichero, podemos lanzarlo como un comando más, ejecutando estos comandos en forma secuencial, o con bifurcaciones condicionales y bucles. Estas características, junto con otras más que posee la Programación Shell, le dan una potencia y utilidad muy superiores al modo de trabajo puramente interactivo. En esta práctica nos centraremos en el trabajo con guiones *shell*.

De entre las distintas variantes de intérpretes *shell*, nosotros utilizaremos la *shell* **Bash**. El nombre proviene de Bourne Again Shell (Otra Shell Bourne), y nació como uno de los proyectos GNU de la FSF (*Free Software Foundation*). La filosofía general de dicho organismo es desarrollar software completamente gratuito y de libre distribución, bajo licencia GNU, filosofía igualmente aplicada a la *shell* Bash.

La *shell* Bash está diseñada para cumplir el estándar POSIX.2 igual que la *shell* Korn, pero además presenta características especiales tomadas de C *shell*. En cualquier caso, si se desea tener una *shell* que sólo sea POSIX, Bash proporciona un *flag* (`--Posix`) que permite ejecutar Bash ateniéndose exclusivamente a dicho estándar.

De la documentación de Bash, podemos extraer algunas de las características aportadas:

- Edición de línea de comandos amigable, personalizable y con completado de nombres.
- Histórico de comandos con mejoras como selección de comandos guardados.
- *Arrays* indexados de tamaño ilimitado.
- Aritmética en bases desde 2 a 16.
- Entrecomillado estándar ANSI-C.
- Expansión de corchetes (similar a C Shell).
- Operaciones con subcadenas en variables.
- Opciones de comportamiento.
- *Prompt* mejorado con caracteres especiales.
- Ayuda integrada.
- Modo POSIX.
- *Shell* restringida.
- Temporización de comandos.
- Pila de directorios.
- Control de órdenes internas.

Bash es una de las *shells* más usadas hoy en día, principalmente porque ha sido adoptada como *shell* predeterminada en las distribuciones de Linux. La característica de ser un proyecto GNU le da el atractivo de ser totalmente libre, por lo que también puede ser usada en cualquier otra distribución UNIX; sólo es necesario bajar el código y compilarlo, o buscar el ejecutable ya compilado para nuestro sistema. Este hecho, junto con la gran cantidad de mejoras que se han ido aportando, la convierte, quizá, en una de las mejores implementaciones POSIX de las *shells* y, además, incluye características adicionales que aún no cumplen dicho estándar.

1.2. Órdenes internas y externas

Cuando tecleamos una orden para que sea ejecutada por la *shell*, generalmente ésta busca en los directorios listados en la variable de entorno `$PATH` un fichero ejecutable con el nombre de la orden en cuestión, a no ser que se especifique la ruta completa a la orden dentro del sistema ficheros:

Mostrar la lista de directorios de búsqueda (se almacenan separados por ':'):

```
$ echo $PATH
/home/juan/bin:/usr/local/bin:/usr/bin:/bin
```

Busca el ejecutable ls (lo encuentra en /bin/ls):

```
$ ls
```

Especificamos la ubicación del ejecutable:

```
$ ./miscript
```

Estas son las llamadas *órdenes externas*.

La *shell* dispone, sin embargo, de una serie de *órdenes internas* (*builtins*) que puede ejecutar directamente sin necesidad de buscar el ejecutable, pues ya tiene incluida esa funcionalidad en su propio código. Estas son algunas de las más habituales:

source `guion`
: La orden `source` o `.` (punto) lee las órdenes contenidas en `guión` y las ejecuta dentro del entorno actual. No es necesario que `guión` tenga permisos de ejecución. Es una manera práctica de definir variables y funciones de manera genérica.

export `[variable]`
: Permite exportar el valor de una variable al entorno de *subshells* generadas a partir de la actual.

builtin `orden`

Ejecuta la orden interna `orden`. Es útil cuando existe una orden externa con el mismo nombre que una interna, como por ejemplo `echo` y `test`.

cd `[dir]`
Cambia al directorio `dir`. Si se ejecuta sin argumentos, vuelve al directorio `home` del usuario.

pwd
Muestra el directorio actual (de *print working directory*).

echo
Escribe sus argumentos por la salida estándar. Acepta una serie de opciones, como por ejemplo `-n`, y `-e`. Con la opción `-n` se evita que escriba un salto de línea al final y con la opción `-e` se permite que interprete caracteres especiales precedidos por `\` como `\t` (tabulador), `\n` (salto de línea), etc.

read `variable`
Lee de la entrada estándar y lo guarda en `variable`.

shift
Desplaza a la izquierda los parámetros posicionales. Se verá su uso más adelante.

exit `[valor]`
Sale de un guión *shell* y devuelve `valor` en la variable de *status*

history
Muestra el histórico de comandos. La *shell* Bash posee una función de histórico de comandos muy potente que nos permite recuperar comandos introducidos anteriormente, modificarlos, editarlos, etc.

jobs
Nos muestra los procesos que se encuentran en ejecución en segundo plano (*background*). Se verá más adelante cuando se estudie la ejecución de procesos en segundo plano.

bg fg
La orden `bg` (de *background*) nos permite pasar a segundo plano un proceso que se encuentra suspendido, por ejemplo porque se le ha enviado la señal correspondiente mediante la combinación de teclas `Ctrl+Z`. La orden `fg [tid]` permite traer a primer plano (*foreground*) un proceso que se encontraba en *background*. Ejemplo:
Se lanza un proceso en primer plano:

```
$ xeyes
```

Se suspende el proceso mediante la combinación `Ctrl+Z`:

```
^Z
[2]+  Stopped                 xeyes
```

El proceso está parado, los ojos no siguen al ratón. Se pasa el proceso a segundo plano y continúa en ejecución, ahora los ojos siguen al ratón de nuevo:

```
$ bg
[2]+ xeyes &
```

Se pasa el proceso a primer plano de nuevo:

```
$ fg 2
xeyes
```

Se termina el proceso mediante la combinación `Ctrl+C`:

```
^C
```

kill `pid|%tid`
Se envía una señal al proceso especificado por `pid` (*process id*) o por `%tid` (*task id*). Se estudiará más adelante.

test [
Evalúa una condición. Se verá con detalle al hablar de las sentencias condicionales.

1.3. Variables y aritmética

Para facilitar la programación de guiones *shell*, Bash permite la definición de variables cuyo valor puede ser recuperado más tarde. Por ejemplo:

```
$ ancho=24
$ echo $ancho
24
$ saludo="Hola, \textquestiondowncomo estas?"
$ echo $saludo
Hola, \textquestiondowncomo estas?
$
```

Es importante tener en cuenta que al asignar un valor a una variable no deben aparecer espacios alrededor del símbolo =, y resulta un error en caso contrario, pues entonces la *shell* considera que el nombre de la variable es una orden que debe ejecutar con una serie de argumentos:

```
$ ancho = 24
bash: ancho: command not found
```

En este caso Bash considera que `ancho` es una orden y los argumentos son "`=`" y "`24`".

Una variable es siempre una cadena de caracteres. Sin embargo, se pueden realizar operaciones aritméticas sobre ellas mediante el operador `$((expresión))`. Por ejemplo:

```
$ ancho=24
$ echo $ancho
24
$ ancho=$(($ancho+1))
$ echo $ancho
25
```

1.4. Redirecciones y cauces

Todo proceso Unix tiene asociados de manera predeterminada 3 descriptores de ficheros: la entrada estándar (0), la salida estándar (1) y la salida de error (2). A la salida de error se dirigen los mensajes provocados por algún error en la ejecución del proceso.

La salida estándar de una orden puede redirigirse a un fichero mediante el operador `>fichero`. Por ejemplo:

```
$ ls -l /etc > sal
$ cat sal
total 1405
drwxr-xr-x  8 root   root  1864 dic 23 10:47 acpi
-rw-r--r--  1 root   root  2975 jun 28  2008 adduser.conf
-rw-r--r--  1 root   root    46 ene 30 11:13 adjtime
-rw-r--r--  1 root   root   196 jun 28  2008 aliases
drwxr-xr-x  3 root   root    88 jun 28  2008 alsa
drwxr-xr-x  2 root   root  7584 ene 26 11:37 alternatives
-rw-r--r--  1 root   root   395 mar  9  2008 anacrontab
-rw-r--r--  1 root   root  4185 jun  9  2008 analog.cfg
```

```
. . .
$
```

Con el operador `>fichero` se trunca el fichero designado. Si queremos añadir el resultado de una orden al contenido de un fichero existente, debemos utilizar el operador de doble redirección `>>fichero`. Por ejemplo:

```
$ echo hola >saludo.txt
$ echo adios >>saludo.txt
$ cat saludo.txt
hola
adios
```

1.4.1. Redirección de la salida de error

También se puede redirigir la salida de error mediante el operador de redirección `2>fichero`. Por ejemplo:

```
$ ls -l /bin/bash /bin/noexiste
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio
```
(salida de error)
-rwxr-xr-x 1 root root 700492 may 12 2008 /bin/bash **(salida estándar)**
$ ls -l /bin/bash /bin/noexiste >/dev/null **(se redirige la salida estándar)**
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio **(salida de error)**
$ ls -l /bin/bash /bin/noexiste 2>/dev/null **(se redirige la salida de error)**
-rwxr-xr-x 1 root root 700492 may 12 2008 /bin/bash **(salida estándar)**

Vemos que podemos redirigir la salida de error indicando el número de descriptor de fichero justo antes del operador de redirección. También podemos redirigir la salida de error a la salida estándar, para que al redirigir esta última aparezcan redirigidos tanto los mensajes de error como la salida normal, mediante el operador `>&`:

```
$ ls -l /bin/bash /bin/noexiste >sal 2>&1
$ cat sal
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio
-rwxr-xr-x 1 root root 700492 may 12  2008 /bin/bash
```

Con la redirección `2>&1` estamos **duplicando** la salida del descriptor 2 (salida de error) sobre el descriptor 1 (salida estándar). Como la salida estándar la hemos redirigido al fichero `sal`, tanto el resultado de la orden `ls` como el mensaje de error van a parar a dicho fichero. Es importante destacar el orden en que se efectúan las redirecciones: primero se redirige la salida estándar y luego se redirige la salida de error a la estándar (que ya está redirigida al fichero). Cuando las redirecciones no se efectúan en el orden correcto, el resultado no será el esperado:

```
$ ls -l /bin/bash /bin/noexiste 2>&1 >sal
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio
$ cat sal
-rwxr-xr-x 1 root root 700492 may 12  2008 /bin/bash
```

Podemos redirigir ambas salidas por separado:

```
$ ls -l /bin/bash /bin/noexiste >ls.out 2>ls.err
$ cat ls.out
-rwxr-xr-x 1 root root 700492 may 12  2008 /bin/bash
$ cat ls.err
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio
```

1.4.2. Cauces

La salida estándar de una orden puede servir también como entrada estándar para la siguiente. Ambas órdenes se encadenan mediante el operador | (cauce o *pipe*). Por ejemplo:

```
$ ls -l /etc | sort | more
drwxr-s---  2 root   dip     96 dic 15 12:53 chatscripts
drwxr-sr-x  2 root   bind   352 ene 16 15:15 bind
drwxr-xr-x 12 root   root   608 oct 20 11:41 X11
drwxr-xr-x 13 root   root   352 nov 20 11:20 texmf
drwxr-xr-x  2 root   root   104 ago  9 15:17 libgda-3.0
drwxr-xr-x  2 root   root   104 nov  4 17:00 vim
. . .
$
```

Se pueden combinar cauces y redirecciones:

```
$ ps u | grep -v ssh >ps.out
$ more ps.out
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
juan      4186  0.0  0.1   6500  3504 pts/0    Ss   09:34   0:00 /bin/bash
juan      6904  0.0  0.1   6500  3532 pts/6    Ss+  14:44   0:00 /bin/bash
juan      6991  0.0  0.1   6500  3532 pts/7    Rs   14:57   0:00 /bin/bash
juan      7042  0.4  1.0  36512 21080 pts/7    S    14:58   0:02 kdvi guion.dvi
juan      7110  0.0  0.0   3924   964 pts/7    R+   15:10   0:00 ps u
juan      7111  0.0  0.0  29040   692 pts/7    S+   15:10   0:00 sort
```

Es muy importante tener en cuenta que las redirecciones de salida se realizan antes de ejecutar las órdenes, por lo que si hacemos algo como lo que sigue:

```
$ sort <ps.out | uniq >ps.out
$ cat ps.out
$
```

acabaremos con un fichero de salida vacío.

Cuestión: ¿ Por qué?

1.5. Listas de órdenes

Cuando la *shell* ejecuta una orden, en la variable especial `$?` se guarda el valor de retorno de la orden, que se corresponde con el parámetro de la función `exit()` invocada al terminar el proceso. Por convenio, el valor de retorno **0** se interpreta como **éxito** o `true`, mientras que un

valor **distinto de 0** se considera como **no-éxito** o `false`. Esta variable `$?` se actualiza continuamente con el valor de retorno del último comando ejecutado.

Una lista de órdenes es una secuencia de comandos (con redirección o no) separados por los operadores **;**, **&**, **&&** o `||`. En una lista de órdenes separadas por el operador **;**, la *shell* ejecuta las órdenes de manera secuencial, esperando que se complete cada uno de los comandos. El código de retorno (el valor de la variable `$?`) es el de la última orden ejecutada.

Si el operador es **&**, el comando se ejecuta en segundo plano, la *shell* no espera a que termine y el código de retorno es 0.

Los operadores **&&** y `||` son operadores AND y OR respectivamente. Por ejemplo, en la lista

```
orden1 && orden2
```

sólo se ejecuta `orden2` si `orden1` retornó con código 0. En cambio, en la lista

```
orden1 || orden2
```

sólo se ejecuta `orden2` si `orden1` retornó con código distinto de 0.

Ejemplos:

```
$ ls /bin/bash && echo Existe ; echo $?
/bin/bash
Existe
0
$ ls /bin/noexiste && echo existe ; echo $?
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio
2
$ ls /bin/bash || echo No existe ; echo $?
/bin/bash
0
$ ls /bin/noexiste || echo No existe ; echo $?
ls: no se puede acceder a /bin/noexiste:
No existe el fichero o el directorio
No existe
0
```

Cuestión: ¿ Por qué en el último ejemplo el código de retorno (el valor de la variable `$?`) es 0, a pesar de que la orden `ls` falló y devolvió un código 2?

1.5.1. Ejecución en segundo plano

Normalmente, los comandos se ejecutan de manera interactiva (*foreground*), de forma que la *shell* interpreta la orden, la ejecuta, nos muestra la salida de la orden, y no nos devuelve el *prompt* hasta que no acaba de ejecutarse el proceso. Cuando queremos dejar un programa en ejecución de manera no interactiva o en segundo plano (*background*) se emplea el operador `&`. Por ejemplo:

```
$ xlogo &
[2] 7584
$
```

El primer número, en este caso el `[2]`, es el número de tarea (*job*), mientras que el segundo número es el identificador de proceso, `PID`. Una vez que el proceso ha iniciado la ejecución como un proceso separado —en realidad, es un proceso hijo de la *shell*—, ya no podemos interrumpirlo mediante la combinación `Ctrl-C`. Para terminar su ejecución hemos de enviarle la señal de terminación (`SIGTERM`) mediante la orden `kill`, indicando como parámetro bien el número de PID, bien el número de tarea precedido por el carácter `%`

```
$ kill 7584
$
[2]+  Terminado               xlogo
$
```

o bien

```
$ kill %2
```

La orden `kill` puede enviar otras señales aparte de la terminación. Puede verse el listado completo mediante `kill -l`.

Para consultar cuáles son los procesos que se están ejecutando actualmente en segundo plano puede emplearse la orden `jobs`:

```
$ jobs
[1]-  Running                 kile shell.tex &
[2]+  Running                 xlogo &
```

Cuando un proceso que se estaba ejecutando en *background* finaliza, la *shell* muestra un mensaje indicando tal circunstancia.

Cuestión: ¿ Qué significa el − y el + que encontramos al lado del [`tid`]?

1.6. Expansión de comodines

Muchas veces debemos referirnos a un conjunto de ficheros que tienen características comunes en sus nombres. Por ejemplo, supongamos que deseamos mostrar en pantalla (mediante `cat`) todos los ficheros que contengan código fuente C. La característica común es que los nombres de los ficheros acaban en `.c`. Esto lo podemos hacer mediante la orden:

```
$ cat *.c
```

En este ejemplo, el carácter `*` concuerda con cualquier cadena de caracteres. De la misma manera, el caráter `?` concuerda con cualquier carácter, y `[conjunto]` concuerda con cualquier carácter dentro de `conjunto`. En otro ejemplo, si un directorio contiene los ficheros llamados `a`, `abc`, `aaaa`, `zba` y `zca`, entonces:

```
$ ls a*
a abc aaaa
$ ls *a
a aaa zba zca
$ ls ?a*
aaaa
$ ls ?[ab]*
aaaa abc zba
```

1.7. Expansión de órdenes

Imaginemos que deseamos contar el número de ficheros cuyo nombre comienza por la letra **a** dentro de un directorio. Podríamos conseguirlo con la orden:

```
$ ls a* | wc -w
13
```

Pero ¿ qué sucede si ahora debemos utilizar el resultado de la orden anterior? La shell nos ha mostrado el resultado por el terminal, pero ya no lo podemos recuperar. La solución es guardar el resultado de la ejecución de una orden dentro de una variable. Esto lo hacemos mediante la expansión de órdenes `$(orden)`:

```
$ num=$( ls a* | wc -w )
$ echo $num
13
```

1.8. Guiones Shell

Un **guión *shell*** es una secuencia de órdenes que se ejecutan como si fuese un programa. El contenido de un guión *shell* se podría ejecutar de manera interactiva, esto es, tecleando una tras otra las órdenes que forman el guión, pero lo más habitual es crear un archivo de texto que contiene la secuencia. Una vez creado el archivo mediante cualquier editor de textos, puede ser invocada su ejecución como si se tratase de una orden más. Sin embargo para ello es necesario que el archivo cumpla con una serie de requisitos, como que posea permisos de ejecución, entre otros. Por ejemplo, analicemos el siguiente caso. Este guión comprueba si la aplicación `xlogo` se encuentra en ejecución; si no lo está, la ejecuta en segundo plano.

```
#! /bin/bash
# Este guion comprueba si xlogo esta en ejecucion y, si no lo esta, la lanza.
ps | grep " xlogo" >/dev/null
# grep devuelve 0 solo si encontro el patron
estaxlogo=$?
# Si estaxlogo == 0, hay al menos un xlogo en ejecucion.
# Si vale != 0, no hay ningun xlogo
if [ $estaxlogo = 0 ] ; then
        echo "Ya esta xlogo en ejecucion."
  else
        xlogo &
fi
exit 0
```

Los comentarios comienzan por el carácter `#`. Un comentario puede empezar en cualquier posición de la línea, y se considera que continúa hasta el final de la misma. Sin embargo, es buena costumbre que el carácter `#` sea el primero de la línea. El caso de `#! /bin/bash` es especial, los caracteres `#!` indican a la *shell* cuál es el programa encargado de interpretar el guión, y deben aparecer siempre en la primera columna de la primera línea del guión. En este caso es la propia `bash` la encargada de interpretar el guión. Un guión *shell* puede hacer referencia a cualquier orden unix disponible, incluso puede invocar la ejecución de otros guiones.

La orden `exit` garantiza que el guión devuelve un código de retorno apropiado. En este caso devuelve el valor 0, que, como ya se dijo, por convenio significa "éxito". Este es el código que se puede inspeccionar inmediatamente después a través de la varible especial `$?`.

Ejercicio: Crear el guión anterior mediante un editor de texto (`kate`, `gedit`, `kwrite`, `vi`, ...) y con el nombre `testxlogo.sh`. Darle permisos de ejecución con `chmod +x testxlogo.sh` e invocarlo varias veces como `./testxlogo.sh`.

Cuestión: ¿ Qué diferencia hay entre ejecutar el guión con `./testxlogo.sh` y `. testxlogo.sh`?

1.8.1. Argumentos

Para que una orden sea más versátil y potente, es necesario que pueda ejecutarse en función de parámetros y opciones que se le pasen de manera externa. Los guiones *shell* no son una excepción, y les podemos pasar parámetros desde la línea de órdenes. Necesitamos algún mecanismo para referenciar los parámetros desde el interior del guión. En *shell*, los parámetros se denotan por `$1`, `$2`, `$3`, ... Estos son los parámetros *posicionales*. `$1` hace referencia al primer parámetro en la línea de órdenes, `$2` al segundo, y así sucesivamente. El parámetro `$0` denota el nombre del propio guión. Por ejemplo, dado el siguiente guión, de nombre `param.sh`:

```
#! /bin/bash
# Ilustracion del paso de parametros
echo "Mi nombre es: $0"
echo "Primer parametro : $1"
echo "Segundo parametro: $2"
exit 0
```

y ahora lo invocamos:

```
$ ./param.sh uno dos
Mi nombre es: ./param.sh
Primer parametro : uno
Segundo parametro: dos
```

Podemos saber cuántos parámetros se han pasado desde la línea de órdenes mediante la variable `$#`. Modificamos el guión anterior:

```
#! /bin/bash
# Ilustracion del paso de parametros
echo "Numero de parametros: $#"
echo "Mi nombre es: $0"
echo "Primer parametro : $1"
echo "Segundo parametro: $2"
exit 0
```

y lo invocamos

```
$ ./param.sh uno dos tres
Numero de parametros: 3
Mi nombre es: ./param.sh
Primer parametro : uno
Segundo parametro: dos
```

Esto nos permitirá inspeccionar un número variable de parámetros y comprobar, por ejemplo, que el número de parámetros es el esperado. Con la orden `shift` se podrán construir guiones que acepten un número variable de parámetros. Se estudiará en la sección 1.8.3 al hablar de los bucles.

1.8.2. Sentencias condicionales

Es fundamental que podamos ejecutar ciertas partes del guión sólo si se cumplen ciertas condiciones. Para ello, usaremos las condiciones y la estructura **if-then-else-fi**. Las condiciones se evalúan mediante la orden `test`. Esta orden devuelve 0 (verdadero) si la condición se cumple, y 1 (falso) si la condición no se cumple. Por ejemplo:

```
$ test -f /bin/bash ; echo $?
0
$ test -f /bin/noexiste ; echo $?
1
```

La orden `test` también se puede escribir como `[`, en cuyo caso se cierra la condición con `]`. Además, puesto que es una orden, debe ir seguida de un espacio:

```
$ [ -f /bin/bash ]; echo $?
0
$ [ -f /bin/noexiste ]; echo $?
1
```

Nota: El convenio que se utiliza en programación *shell* es el contrario al utilizado en otros lenguajes de programación, como por ejemplo C. Aquí, el código 0 significa "verdadero" y algo distinto de 0 significa "falso". De esta forma podemos distinguir, asignando distintos códigos de salida, entre las diferentes condiciones de error que se han podido producir.

Con la estructura `if-then-else-fi` podemos ejecutar bloques de órdenes cuando una determinada condición es verdadera (bloque `then`) o cuando es falsa (bloque `else`). Por ejemplo:

```
if [ -f /bin/bash ] ; then
        echo "El archivo /bin/bash existe y es regular"
   else
        echo "El archivo /bin/bash no existe o no es regular"
fi
```

Las condiciones que podemos examinar con `test` son de tres tipos: comparaciones de cadenas, comparaciones aritméticas y condiciones de ficheros. Se puede ver un resumen en la tabla 1.1. Hay que resaltar que los operadores de comparación de cadenas deben ir precedidos y seguidos por sendos espacios. Además, el comparador de igualdad es =, y no ==. Éste último puede emplearse en Bash, pero tiene un significado diferente.

Ejemplo: El siguiente guión lee una respuesta del usuario y actúa en consecuencia. Se usa la construcción **elif**, que permite añadir una segunda condición al bloque `else` (`elif` es abreviatura de `else if`):

```
#! /bin/bash
echo "\textquestiondownEs por la ma\~nana? Por favor, responda si o no."
read tiempo
if [ $tiempo = "si" ] ; then
        echo "Buenos dias."
  elif [ $tiempo = "no" ] ; then
        echo "Buenas tardes."
```

Comparación de cadenas	**Resultado**
`cadena1 = cadena2`	Verdadero si las cadenas son iguales
`cadena1 != cadena2`	Verdadero si las cadenas no son iguales
`-n cadena`	Verdadero si la cadena es no nula
`-z cadena`	Verdadero si la cadena es nula (vacía)
Comparación aritmética	**Resultado**
`expresión1 -eq expresión2`	Verdadero si ambas expresiones son iguales
`expresión1 -ne expresión2`	Verdadero si ambas expresiones no son iguales
`expresión1 -gt expresión2`	Verdadero si `expresión1` $>$ `expresión2`
`expresión1 -ge expresión2`	Verdadero si `expresión1` $\geq$ `expresión2`
`expresión1 -lt expresión2`	Verdadero si `expresión1` $<$ `expresión2`
`expresión1 -le expresión2`	Verdadero si `expresión1` $\leq$ `expresión2`
`! expresión1`	Verdadero si `expresión1` es falsa
Condición de fichero	**Resultado**
`-d fichero`	Verdadero si `fichero` es un directorio
`-e fichero`	Verdadero si `fichero` existe
`-f fichero`	Verdadero si `fichero` es un fichero regular
`-r fichero`	Verdadero si `fichero` tiene permisos de lectura
`-s fichero`	Verdadero si `fichero` tiene longitud > 0
`-w fichero`	Verdadero si `fichero` tiene permisos de escritura
`-x fichero`	Verdadero si `fichero` tiene permisos de ejecución

Cuadro 1.1: Condiciones de `test`

```
  else
        echo "Lo siento, no he entendido $tiempo."
        echo "Por favor, responda si o no."
        exit 1
fi
exit 0
```

Ejercicio: Crear el guión anterior y ejecutarlo probando a introducir distintas respuestas. Probar a teclear INTRO (introduciendo, así una cadena vacía en la variable `tiempo`):

```
bash: [: =: unary operator expected
```

En este caso la ejecución falla. ¿ Por qué?. Cuando la variable `tiempo` toma como valor la cadena vacía, la sustitución que hace la *shell* en la comparación es:

```
[ = "si" ]
```

y falla la ejecución porque el operador = espera comparar dos cadenas, pero sólo tiene la cadena "`si`". Para solucionarlo, podemos entrecomillar la variable `tiempo`:

```
if [ "$tiempo" = "si" ] ; then
...
  elif [ "$tiempo" = "no" ] ; then
...
```

y ahora la condición, en caso de que introduzcamos la cadena vacía, será:

```
if [ "" = "si" ] ; then
```

lo que es correcto desde el punto de vista sintáctico.

Cuestión: ¿ Qué cámbios habría que hacer al script para que admisiese "`Si`" y '"`No`" como contestaciones válidas?

1.8.3. Bucles

Cuando queremos iterar a través de un rango de valores, podemos usar un bucle. Las estructuras de bucles que nos permite `Bash` son **for**, **while** y **until**. Estudiaremos con más detalles los dos primeros.

Bucle `for`

La sintaxis de un bucle `for` es la siguiente:

```
for variable in valores ; do
    ordenes
done
```

Por ejemplo:

```
for i in hola adios "hasta luego" ; do
  echo $i
done
```

daría como resultado:

```
hola
adios
hasta luego
```

Nótese que, al encerrar entre comillas las palabras "`hasta luego`", se considera una sola palabra. La *shell* nos ofrece una característica que es el **entrecomillado**. Al encerrar una cadena de caracteres entre comillas, se agrupa toda la cadena para formar una sola palabra, independientemente del número de espacios en blanco que contenga. Así, la cadena de caracteres "`hasta luego`", se ha agrupado en una sola palabra que se ha pasado, en el ejemplo anterior, a la variable `$i` como un todo.

Cuando utilizamos comillas dobles ("), se permite, además, la expasión de variables. En cambio, cuando utilizamos comillas simples (’), no se realiza dicha expansión, sino que la cadena de caracteres se considera literal. Véase el siguiente ejemplo:

```
hl="hasta luego"
for i in hola adios $hl ; do
  echo $i
done
```

da como resultado:

```
hola
adios
hasta
luego
```

En cambio:

```
hl="hasta luego"
for i in hola adios "$hl" ; do
  echo $i
done
```

da como resultado:

```
hola
adios
hasta luego
```

Y por último:

```
hl="hasta luego"
for i in hola adios '$hl' ; do
  echo $i
done
```

da como resultado:

```
hola
adios
$hl
```

En el bucle `for`, el rango de `valores` puede resultar de la expansión de una variable o de comodines:

```
for i in * ; do
  echo "Fichero $i"
done
```

Ejercicio: Comprobar en este caso qué sucede si escribimos "`*`" (con comillas dobles) o '`*`' (con comillas simples).

Cuando queremos iterar sobre un rango de valores numéricos, podemos utilizar el bucle `for` con una sintaxis parecida al lenguaje C:

```
for (( i=0 ; i<=15 ; i++ )) ; do
  echo $i
done
```

Resulta:

```
0
1
...
15
```

Esto es más fácil que escribir (aunque también es correcto):

```
for i in 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ; do
  echo $i
done
```

Bucle `while`

La sintaxis de un bucle `while` es como sigue:

```
while condicion ; do
    ordenes
done
```

Por ejemplo:

```
#! /bin/bash
echo "Introduzca la contrase\~na:"
read passwd
while [ "$passwd" != "secreto" ] ; do
    echo "Lo siento, intentelo de nuevo."
    read passwd
done
exit 0
```

En la sección 1.8.1 se vio cómo podemos pasar parámetros al guión *shell*, y también se indicó que `$#` contiene el número total de parámetros. Podemos utilizar la orden interna `shift` para hacer guiones que acepten un número variable de argumentos. Por ejemplo:

```
#! /bin/bash
# Guion que cuenta el numero de parametros y los muestra uno a uno
echo "Numero de parametros: $#"
echo "Mi nombre es: $0"
while [ $# -gt 0 ] ; do
    echo "Numero de parametros restantes: $#"
    echo "Parametro : $1"
    shift
done
exit 0
```

Un ejemplo de su ejecución:

```
$ ./argcount.sh arbol casa perro
Numero de parametros: 3
Mi nombre es: ./argcount.sh
Numero de parametros restantes: 3
Parametro : arbol
Numero de parametros restantes: 2
Parametro : casa
Numero de parametros restantes: 1
Parametro : perro
```

La orden `shift` "desplaza" hacia la izquierda los parámetros posicionales, haciendo que el que era `$2` pase a ser `$1`, el que era `$3` pase a ser `$2`, y así sucesivamente. Además, decrementa en 1 el valor de `$#`.

Bucle `until`

El bucle `until` es muy parecido al bucle `while`, pero con la condición opuesta. Su sintaxis es la siguiente:

```
until condicion ; do
    ordenes
done
```

1.9. Órdenes útiles

A continuación veremos algunas órdenes útiles para la creación de guiones *shell*. De todas ellas puede consultarse las opciones posibles a través de la página de manual `man orden`.

1.9.1. grep

La orden `grep` busca en la entrada, bien en la que se le especifica con nombres de ficheros o bien en la entrada estándar si no se le dan dichos nombres o si uno de éstos consiste en –, líneas que concuerden con el patrón dado. Si no se dice otra cosa, `grep` muestra las líneas que concuerden.

Existen tres grandes variantes de `grep`, controladas por las siguientes opciones:

-G, --basic-regexp Interpreta patrón como una expresión regular básica (ver más adelante). Éste es el comportamiento predeterminado.

-E, --extended-regexp Interpreta patrón coma una expresión regular extendida (ver más adelante). Es equivalente a invocar la orden como `egrep`.

-F, --fixed-strings Interpreta patrón como una lista de cadenas de caracteres fijas, separadas por saltos de línea; se busca la concordancia de una cualquiera de ellas.

Estas son algunas de las opciones que entiende `grep`:

-c, --count Suprime la salida normal; en su lugar muestra el número de líneas que concuerdan con el patrón para cada fichero de entrada. Con la opción -v, --invert-match (ver más abajo), muestra el número de líneas que no concuerden.

-e patrón, --regexp=PATRON Emplea patrón como el patrón; útil para proteger patrones que comiencen con -.

-f fichero, --file=FICHERO Obtiene el patrón de `fichero`.

-h, --no-filename Suprime la impresión de los nombres de ficheros antes de las líneas concordantes en la salida, cuando se busca en varios ficheros.

-i, --ignore-case No hace caso de si las letras son mayúsculas o minúsculas ni en el patrón ni en los ficheros de entrada.

-L, --files-without-match Suprime la salida normal; en su lugar muestra el nombre de cada fichero de entrada donde no se encuentre ninguna concordancia y por lo tanto de cada fichero que no produciría ninguna salida. La búsqueda se detendrá al llegar a la primera concordancia.

-l, --files-with-matches Suprime la salida normal; en su lugar muestra el nombre de cada fichero de entrada que produciría alguna salida. La búsqueda se detendrá en la primera concordancia.

-n, --line-number Prefija cada línea de salida con el número de línea de su fichero de entrada correspondiente.

-q, --quiet Silencioso; suprime la salida normal. La búsqueda finaliza en la primera concordancia.

-s, --silent Suprime los mensajes de error sobre ficheros que no existen o no se pueden leer.
-v, --invert-match Invierte el sentido de la concordancia, para seleccionar las líneas donde no se encuentra el patrón.
-w, --word-regexp Selecciona sólo aquellas líneas que contienen concordancias que forman palabras completas. La comprobación consiste en que la cadena de caracteres concordante debe estar al principio de la línea o precedida por un carácter que no forme parte de una palabra. De forma similar, debe estar o al final de la línea o seguida por un carácter no constituyente de palabra. Los caracteres que se consideran como parte de palabras son letras, dígitos y el subrayado.
-x, --line-regexp Selecciona sólo aquellas concordancias que constan de toda la línea.

Expresiones Regulares

Una expresión regular es un patrón que describe un conjunto de cadenas de caracteres. Las expresiones regulares se construyen de forma análoga a las expresiones aritméticas, combinando expresiones más pequeñas mediante ciertos operadores para formar expresiones complejas.

El programa grep entiende dos versiones diferentes de sintaxis para las expresiones regulares: la "básica" y la "extendida". En la versión de `grep` de GNU, no hay diferencia en usar una u otra en cuanto a la funcionalidad disponible. En otras implementaciones, las expresiones regulares básicas son menos potentes. La siguiente descripción se aplica a expresiones regulares extendidas; las diferencias con las básicas se resumen a continuación.

Los bloques de construcción fundamentales son las expresiones regulares que concuerdan con un solo carácter. La mayoría de los caracteres, incluyendo todas las letras y dígitos, son expresiones regulares que concuerdan consigo mismos. Cualquier meta-carácter con un significado especial debe ser protegido precediéndolo con una barra inclinada inversa.

Una lista de caracteres rodeados por `[` y `]` concuerda con cualquier carácter de esa lista; si el primer carácter de la lista es el acento circunflejo ^ entonces concuerda con cualquier carácter de fuera de la lista. Por ejemplo, la expresión regular `[0123456789]` concuerda con cualquier carácter dígito. Se puede especificar un rango de caracteres ASCII dando el primero y el último, separados por un guión. Finalmente, están predefinidas ciertas clases de caracteres, con un nombre para cada una. Estos nombres son auto-explicativos (consultar `man wctype` en caso contrario), y son `[:alnum:]`, `[:alpha:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]`, y `[:xdigit:]`. Por ejemplo, [[:alnum:]] significa (en inglés) [0-9A-Za-z], salvo que la última forma depende de que la codificación de caracteres siga el estándar ISO-646 o ASCII, mientras que la primera es transportable. Debe observarse que los corchetes en estos nombres de clases son parte de los nombres simbólicos, y deben incluirse además de los corchetes que delimitan la lista entre corchetes. La mayor parte de los meta-caracteres pierden su significado especial dentro de estas listas. Para incluir un `]` literal, debe aparecer el primero de la lista. De forma similar, para incluir un ^ literal, debe figurar en cualquier posición excepto la primera. Finalmente, para incluir un − literal, debe aparecer el último de la lista.

El punto `.` concuerda con cualquier carácter individual.

El acento circunflejo ^ y el signo del dólar $ son meta-caracteres que respectivamente concuerdan con la cadena vacía al comienzo y al final de una línea.

Una expresión regular que concuerde con un solo carácter puede ser seguida por uno de estos operadores de repetición:
? El elemento precedente es opcional y concuerda como mucho una vez.
***** El elemento precedente concordará cero o más veces.
+ El elemento precedente concordará una o más veces.

{n} El elemento precedente concuerda exactamente `n` veces.
{n,} El elemento precedente concuerda `n` o más veces.
{,m} El elemento precedente es opcional y concuerda como mucho `m` veces.
{n,m} El elemento precedente concuerda como poco `n` veces, pero no más de `m` veces.

En las expresiones regulares básicas, los meta-caracteres ?, +, {, |, (, y) pierden su significado especial; en su lugar deben emplearse las versiones protegidas mediante la barra inversa \?, \+, \{, \|, \(, y \).

Diagnósticos

Normalmente, el *status* de salida es 0 si se encuentran concordancias, y 1 si no se encuentran (la opción -v invierte el sentido del *status* de salida). El *status* de salida es 2 si había errores de sintaxis en el patrón, si los ficheros de entrada eran inaccesibles, o en caso de otros errores del sistema.

Ejemplos

```
$ cat prueba
01234
abcdefg
aaabcd
bbccddee
$ grep -E ^a+ prueba
abcdefg
aaabcd
$ grep -E ^a{2} prueba
aaabcd
$ grep -E b+ prueba
abcdefg
aaabcd
bbccddee
$ grep -E [0-9] prueba
01234
```

1.9.2. wc

La orden wc escribe el número de saltos de línea, de palabras o de caracteres en un fichero.

Sinopsis

```
wc [opcion]... [fichero]...
```

Descripción

La orden `wc` escribe la cuenta del número de líneas, palabras o caracteres de un `fichero`. Si se especifican más de un `fichero`, se escribe también el total. Si no se especifica el `fichero`, o el nombre es `-`, entonces se toma la entrada estándar.

-c, --bytes Escribe la cuenta de bytes
-w, --words Escribe la cuenta de palabras
-l, --lines Escribe la cuenta de líneas

Ejemplos

```
$ wc -l /etc/passwd
35 /etc/passwd
$ wc -c /etc/passwd
1628 /etc/passwd
$ wc -w /etc/passwd
51 /etc/passwd
$ wc /etc/passwd
 35  51 1628 /etc/passwd
```

1.9.3. sort

La orden `sort` ordena las líneas de texto de una serie de ficheros

Sinopsis

```
sort [opcion]... [fichero]...
```

Descripción

La orden sort escribe la concatenación ordenada del `fichero(s)` a la salida estándar. Si se omite `fichero`, se toma la entrada estándar. Se reconocen, entre otras, estas opciones de ordenación:

-b, --ignore-leading-blanks Ignora los espacios en blanco al comienzo de la línea

-f, --ignore-case No hace distinción entre letras mayúsculas y minúsculas (las considera todas como mayúsculas)

-g, --general-numeric-sort Realiza las comparaciones según el valor numérico extendido (en notación científica)

-n, --numeric-sort Realiza las comparaciones según el valor numérico de cadena

-r, --reverse Invierte el orden

-k, --key=POS1,POS2 Utiliza como clave de ordenación desde `POS1` a `POS2`, en lugar de comenzar por el principio de la línea

-t, --field-separator=SEP Utilizar `SEP` como separador de campos, en lugar de los espacios en blanco

Ejemplos

```
$ cat nombres
Juan
Pedro
Jacinta
Abel
Rosa
Eva
$ sort nombres
Abel
Eva
Jacinta
Juan
```

```
Pedro
Rosa
$ cat numbers
1000
23
13
123
1.1
1e-1
$ sort numbers
1000
1.1
123
13
1e-1
23
$ sort -g numbers
1e-1
1.1
13
23
123
1000
$ sort -n numbers
1.1
1e-1
13
23
123
1000
```

Observar la diferencia en los resultados entre las tres órdenes `sort numbers`, `sort -g numbers` y `sort -n numbers`. En el primer caso, se realiza una comparación alfanumérica según el código ASCII y teniendo en cuenta que '.'<'0'<'9'<'A'<'Z'<'a'<'z'. En el segundo caso, `sort -g numbers`, la ordenación se realiza según el valor numérico en notación científica, por lo que aparece primero la línea con `1e-1`, porque su valor equivalente es 0,1. En el tercer caso, `sort -n numbers`, la comparación numérica se hace teniendo en cuenta sólo el valor de la cadena de caracteres, pero sin interpretar la notación científica, por lo que primero aparecen los valores `1.1` y `1e-1`, pues ambos son equivalentes a `1`; a partir del primer carácter no numérico se realiza una comparación alfanumérica.

1.9.4. cut

La orden `cut` elimina partes de cada línea de sus archivos de entrada.

Sinopsis

```
cut [opcion]... [fichero]...
```

Descripción

Escribe en la salida estándar partes seleccionadas de cada línea de entrada. Si se omite `fichero`, se toma la entrada estándar.

Algunas de las opciones posibles son:

-c, --characters=LISTA Selecciona sólo estos caracteres
-d, --delimiter=DELIM Utiliza `DELIM` como separador de campos, en lugar de un tabulador
-f, --fields=LISTA Selecciona sólo estos campos; además imprime las líneas que no contengan el carácter separador de campos, salvo que se especifique la opción `-s`
--complement Invierte la selección
-s, --only-delimited No escribe las líneas que no contienen el carácter separador de campos
--output-delimiter=STRING Emplea STRING como separador de campos en la salida; si se omite, se utiliza el mismo separador que en la entrada

Para las opciones `-c` ó `-f` debe especificarse una `LISTA`. Una `LISTA` es uno o más rangos, separados por comas. Los rangos se pueden especificar de las siguientes maneras:

N El carácter o campo N-ésimo
N- Desde el cáracter o campo N-ésimo hasta el final de la línea
N-M Desde el carácter o campo N-ésimo hasta el M-ésimo, ambos inclusive
-N Desde el primer campo o carácter hasta el N-ésimo (incluído)

Ejemplos

```
 $ cat nombres
Gomez Perez, Adela, 8.000
Alonso Corral, Pedro, 5.000
Gutierrez Fernandez, Federico, 6.000
$ cut -c 1,3,5-9,20- nombres
Gmz Per 8.000
Aoso Coo, 5.000
Gterrez, Federico, 6.000
$ cut -d ',' -f 1,3 nombres
Gomez Perez, 8.000
Alonso Corral, 5.000
Gutierrez Fernandez, 6.000
```

1.9.5. Fichero /etc/passwd

De la página de manual (`man 5 passwd`):

Passwd es un fichero de texto que contiene una lista de las cuentas del sistema, proporcionando para cada cuenta cierta información útil como el identificador (ID) de usuario, el ID de grupo, el directorio "`home`", el intérprete de órdenes, etc. Con frecuencia, también contiene la contraseña cifrada de cada cuenta. Este fichero debe tener permiso de lectura para todos (muchas utilidades, como `ls`, lo usan para traducir el número de identificador de usuario (UID) al nombre del usuario), pero sólo el superusuario debe poder escribirlo.

En los buenos viejos tiempos no había grandes problemas con estos permisos generales de lectura. Cualquiera podía leer contraseñas cifradas, ya que el hardware era demasiado lento para descifrar una clave bien elegida y, además, la suposición básica solía ser que la comunidad de usuarios era bastante amigable. Hoy en día, mucha gente utiliza alguna versión del paquete `shadow password`, donde en `/etc/passwd` encontramos *'s en lugar de las claves cifradas; éstas se encuentran en el fichero `/etc/shadow`, el cual sólo lo puede leer el superusuario.

Al crear una nueva cuenta, debe escribirse primero un asterisco en el campo de contraseña y a continuación usar `passwd`(1) para asignarla.

Hay una entrada por línea, cada línea tiene el siguiente formato:

`cuenta:contraseña:UID:GID:GECOS:directorio:intérprete`

Las descripciones de los campos son las siguientes:

cuenta el nombre del usuario en el sistema. No debe contener letras mayúsculas.
contraseña la contraseña cifrada del usuario o un asterisco.
UID el número del ID de usuario.
GID el número del ID de grupo primario para este usuario.
GECOS Este campo es opcional y sólo se usa para propósitos de información. Normalmente, contiene el nombre completo del usuario.
directorio el directorio base del usuario (`$HOME`).
intérprete el programa que se debe ejecutar cuando el usuario ingresa (si está vacío, se utiliza `/bin/sh`). Si se establece a un ejecutable que no existe, el usuario será incapaz de entrar al sistema a través de `login`.

1.9.6. Fichero /etc/group

`/etc/group` es un fichero ASCII que define los grupos a los cuales pertenecen los usuarios del sistema. Hay una entrada por línea, y cada línea tiene el siguiente formato:

`nombre_grupo:contraseña:GID:lista_usuarios`

Las descripciones de los campos son:

nombre_grupo el nombre del grupo.
contraseña la contraseña del grupo (cifrada). Si este campo está vacío, no se utiliza ninguna contraseña.
GID el número de ID del grupo.
lista_usuarios los nombres de usuario de todos los miembros del grupo, separados por comas.

1.10. Ejemplos

1.10.1. Cauces

Esta lista de órdenes muestra en la salida estándar el número de bibliotecas diferentes que existen en el directorio `/usr/lib`, independientemente de que sean estáticas o dinámicas y del número de versión. El listado de las bibliotecas se guarda en el fichero `listalibs`:

```
ls -d /usr/lib/lib* | cut -c 13- | cut -d . -f 1 | sort | uniq | tee listalibs
| wc -l
```

Nota: Ver el cometido de la orden `tee` en la página del manual (con `man tee`).

1.10.2. Búsqueda recursiva

Este par de guiones buscan los archivos creados o modificados en el día de hoy. El primer guión, fut.sh, formatea la fecha actual y llama al segundo guión, recursivo.sh. Éste último realiza un descenso por los directorios que están por debajo del inicial, invocándose a sí mismo de manera recursiva.

```
$ cat fut.sh
#!/bin/bash
# Este guion encuentra los ficheros creados/actualizados hoy
# Utiliza el guion recursivo.sh
#
# Uso: fut.sh <directorio>
ruta=$(dirname $0)
diractual=$PWD
cd $ruta; rutaA=$PWD; cd $diractual
PATH=$rutaA:$PATH
hoy=$(date +%Y-%m-%d)
directorio=$1
if [ -z $directorio ]; then
    directorio=.
fi
recursivo.sh "$directorio" $hoy
```

```
$ cat recursivo.sh
#!/bin/bash
# Este guion encuentra los ficheros creados/actualizados en cierta fecha
# Es utilizado por el guion fut.sh
#
# Uso: recursivo.sh <directorio> <fecha>
cd "$1"
for fichero in *; do
    if [ -d "$fichero" ]; then
        recursivo.sh "$fichero" $2
    fi
    if [ -f "$fichero" ] ;then
        fichero_largo=$(ls -l --time-style="+%Y-%m-%d" "$fichero")
        chequeo=$(echo $fichero_largo | grep $2)
        if [ -n "$chequeo" ]; then
            echo $PWD/$fichero
        fi
    fi
done
```

Ejemplo de uso:

```
$ ./fut.sh .
/home/juan/Documents/Clases/LSO/2009/fut.sh
/home/juan/Documents/Clases/LSO/2009/pr2_2009.odp
/home/juan/Documents/Clases/LSO/2009/recursivo.sh
/home/juan/Documents/Clases/LSO/2009/recursivo.sh~
/home/juan/Documents/Clases/LSO/2009/shell.tex
/home/juan/Documents/Clases/LSO/2009/shell.tex~
```

1.10.3. Control de sesiones

Este guión muestra de manera continua el número de sesiones que tiene abiertas un usuario.

```
$ cat nterm.sh
#!/bin/bash
# Este guion informa del numero de sesiones que tiene abiertas un usuario
# Primero se comprueba que hemos pasado el parametro del nombre de usuario
if [ $# -ne 1 ] ; then
  echo "Uso: $0 nombre_de_usuario"
  exit 1
fi
# Ahora comprueba que el usuario existe en /etc/passwd
grep "^$1:" /etc/passwd >/dev/null
if [ $? -ne 0 ] ; then
  echo "El usuario $1 no existe en el sistema."
  exit 1
fi
# Informa de cuantas sesiones tiene abiertas el usuario
oldnum=$(who | grep "^$1 " |wc -l)
echo "El usuario $1 tiene abiertas $oldnum sesiones"
# Bucle infinito. Se acaba con ^C
while [ 1 ] ; do
  # numero de sesiones actual
  num=$(who | grep "^$1 " |wc -l)
  if [ $num -lt $oldnum ] ; then
    # el numero actual es menor que el antiguo, luego se han cerrado sesiones
    date
    echo "El usuario $1 ha cerrado $(($oldnum-$num)) sesiones ($num actualmente)"
    # Se actualiza en numero de sesiones
    oldnum=$num
  elif [ $num -gt $oldnum ] ; then
    # el usuario ha abierto sesiones
    date
    echo "El usuario $1 ha abierto $(($num-$oldnum)) sesiones ($num actualmente)"
    # Se actualiza en numero de sesiones
    oldnum=$num
  fi

  # Esperamos unos segundos entre actualizaciones
  sleep 2
done
```

Y un ejemplo de ejecución:

```
$ ./nterm.sh usuario_local
El usuario usuario_local tiene abiertas 3 sesiones
mar feb 17 15:37:44 CET 2009
El usuario usuario_local ha abierto 1 sesiones (4 actualmente)
mar feb 17 15:37:56 CET 2009
El usuario usuario_local ha abierto 1 sesiones (5 actualmente)
mar feb 17 15:39:12 CET 2009
El usuario usuario_local ha cerrado 2 sesiones (3 actualmente)
```

1.11. Ejercicio Propuesto

Se debe programar un guión *shell* llamado `quien.sh` que aceptará como parámetro el nombre de un fichero *password* (véase apartado 1.9.5) y que realizará las siguientes tareas:

- Comprobará que el nombre de usuario existe en el fichero de *password*
- Mostrará la siguiente información a partir del archivo:
  - Nombre de usuario
  - Nombre completo
  - Directorio `home`
  - Shell
  - UID y GID
- Cuando el número de parámetros no es correcto, o si el fichero de *password* no existe o no es regular, se mostrará un mensaje por la salida estándar de error y se saldrá del guión con *status* 1 y 2, respectivamente.
- Cuando el usuario introducido no exista en el fichero, se mostrará un mensaje indicativo por la salida estándar de error y se pedirá otro usuario.
- Cuando el usario introducido sea `fin`, el guión terminará su ejecución con *status* 0.

Ejemplo de ejecución:

```
$ ./quien.sh
Uso: ./quien.sh fichero_de_passwd
$ ./quien.sh /etc/pas
El fichero /etc/pas no existe o no es regular.
$ ./quien.sh ./passwd

Introduzca el usuario (fin para acabar): juan
Login: juan              Nombre: Juan Carlos,,,
UID: 1000                GID: 1000
Home: /home/juan         Shell: /bin/bash
Introduzca el usuario (fin para acabar): roo
El usuario roo no existe en ./passwd.
Introduzca el usuario (fin para acabar): root
Login: root              Nombre: root
UID: 0                   GID: 0
Home: /root              Shell: /bin/bash
Introduzca el usuario (fin para acabar): bind
Login: bind              Nombre:
UID: 107                 GID: 112
Home: /var/cache/bind    Shell: /bin/false
Introduzca el usuario (fin para acabar): fin
Gracias
```

1.12. El directorio `/proc`

El siguiente texto forma parte del "Capítulo 7: The `/proc` File System" del libro Advanced Linux Programming (Mark Mitchell, Jeffrey Oldham and Alex Samuel, junio de 2001):

Try invoking the `mount` command without arguments - this displays the file systems currently mounted on your GNU/Linux computer. You'll see one line that looks like this:

```
none on /proc type proc (rw)
```

This is the special `/proc` file system. Notice that the first field, none, indicates that this file system isn't associated with a hardware device such as a disk drive. Instead, `/proc` is a window into the running Linux kernel. Files in the `/proc` file system don't correspond to actual files on a physical device. Instead, they are magic objects that behave like files but provide access to parameters, data structures, and statistics in the kernel. The "contents" of these files are not always fixed blocks of data, as ordinary file contents are. Instead, they are generated on the fly by the Linux kernel when you read from the file.You can also change the configuration of the running kernel by writing to certain files in the `/proc` file system. Let's look at an example:

```
% ls -l /proc/version
-r--r--r-- 1 root root 0 Jan 17 18:09 /proc/version
```

Note that the file size is zero; because the file's contents are generated by the kernel, the concept of file size is not applicable. Also, if you try this command yourself, you'll notice that the modification time on the file is the current time.

What's in this file? The contents of `/proc/version` consist of a string describing the Linux kernel version number. It contains the version information that would be obtained by the `uname` system call, described in Chapter 8, "Linux System Calls", in Section 8.15 `uname`, plus additional information such as the version of the compiler that was used to compile the kernel. You can read from `/proc/version` like you would any other file. For instance, an easy way to display its contents is with the cat command.

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com) (gcc version
egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)) #1 Tue Mar 7
21:07:39 EST 2000
```

The various entries in the `/proc` file system are described extensively in the `/proc` man page (Section 5). To view it, invoke this command:

```
% man 5 proc
```

In this chapter, we'll describe some of t he features of the `/proc` file system that are most likely to be useful to application programmers, and we'll give examples of using them. Some of the features of `/proc` are handy for debugging, too. If you're interested in exactly how `/proc` works, take a look at the source code in the Linux kernel sources, under `/usr/src/linux/fs/proc/`.

Most of the entries in `/proc` provide information formatted to be readable by humans, but the formats are simple enough to be easily parsed. For example, `/proc/cpuinfo` contains information about the system CPU (or CPUs, for a multiprocessor machine). The output is a table of values, one per line, with a description of the value and a colon preceding each value.

A simple way to extract a value from this output is to read the file into a buffer and parse it. Be aware, however, that the names, semantics, and output formats of entries in the `/proc` file system might change in new Linux kernel revisions. If you use them in a program, you should make sure that the program's behavior degrades gracefully if the `/proc` entry is missing or is formatted unexpectedly.

The `/proc` file system contains a directory entry for each process running on the GNU/Linux system. The name of each directory is the process ID of the corresponding process.1 These directories appear and disappear dynamically as processes start and terminate on the system. Each directory contains several entries providing access to information about the running process. From these process directories the `/proc` file system gets its name. Each process directory contains these entries:

- `cmdline`: contains the argument list for the process.
- `cwd`: is a symbolic link that points to the current working directory of the process (as set, for instance, with the `chdir` call).
- `environ`: contains the process's environment.
- `exe`: is a symbolic link that points to the executable image running in the process.
- `fd`: is a subdirectory that contains entries for the file descriptors opened by the process.
- `maps`: displays information about files mapped into the process's address. For each mapped file, maps displays the range of addresses in the process's address space into which the file is mapped, the permissions on these addresses, the name of the file, and other information. The maps table for each process displays the executable running in the process, any loaded shared libraries, and other files that the process has mapped in.
- `root`: is a symbolic link to the root directory for this process. Usually, this is a symbolic link to `/`, the system root directory. The root directory for a process can be changed using the `chroot` call or the `chroot` command.
- `stat`: contains lots of status and statistical information about the process. These are the same data as presented in the status entry, but in raw numerical format, all on a single line. The format is difficult to read but might be more suitable for parsing by programs. If you want to use the stat entry in your programs, see the `proc` man page, which describes its contents, by invoking `man 5 proc`.
- `statm`: contains information about the memory used by the process.
- `status`: contains lots of status and statistical information about the process, formatted to be comprehensible by humans.

Note that for security reasons, the permissions of some entries are set so that only the user who owns the process (or the superuser) can access them.

Bibliografía

- Chet Ramey y Brian Fox. *Bash Reference Manual* Edition 4.1, December 2009.
- N. Matthew y R. Stones. *Beginning Linux Programming*. Wrox, 4ªedición, 2008. Capítulo 2.
- P. López Soto. *Técnicas de Programación de la Shell de Linux para Administradores* Universidad de Málaga.
- GNU Project. *Manual de Bash* http://www.gnu.org/software/bash/manual