Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE I.S.O.

2º de Grado en Ingeniería Informática

Boletín de prácticas 3 – Programación de *shell scripts* en Linux (2/3)

Curso 2012/2013

Índice

1. Entrada/salida estándar y redirección

La filosofía de UNIX/Linux es en extremo modular. Se prefieren las herramientas pequeñas que realizan tareas concretas a las macro-herramientas que realizan todo. Para completar el modelo, es necesario proporcionar el mecanismo para ensamblar estas herramientas en estructuras más complejas. Esto se realiza por medio del redireccionamiento de las entradas y las salidas.

Todos los programas tienen por defecto una entrada estándar (teclado) y dos salidas: la salida estándar (pantalla) y la salida estándar de error (pantalla). En ellos se puede sustituir el dispositivo por defecto por otro dispositivo. Con esto se consigue que los datos de la entrada estándar para un programa se puedan leer de un archivo y los de la salida (estándar o error) se puedan enviar a otro archivo.

La entrada estándar, la salida estándar y la salida estándar de error se asocian a los programas mediante tres descriptores de fichero con los cuales se comunican con otros procesos y con el usuario. Estos tres descriptores son:

- `stdin` (entrada estándar): a través de este descriptor de fichero los programas reciben datos de entrada. Normalmente `stdin` está asociado a la entrada del terminal en la que está corriendo el programa, es decir, al teclado. Cada descriptor de fichero tiene asignado un número con el cual podemos referirnos a él dentro de un script, en el caso de stdin es el 0.
- `stdout` (salida estándar): es el descriptor de fichero en el que se escriben los mensajes que imprime el programa. Normalmente, estos mensajes aparecen en la pantalla para que los lea el usuario. Su descriptor de fichero es el número 1.
- `stderr` (salida estándar de error): es el descriptor de fichero en el que se escriben los mensajes de error que imprime el programa. Normalmente coincide con `stdout`. Tiene como descriptor de fichero el número 2.

Estos tres descriptores de fichero pueden redireccionarse, consiguiendo comunicar unos procesos con otros, de forma que trabajen como una unidad, haciendo cada uno una tarea especializada o, simplemente, almacenando los datos de salida en un fichero determinado o recibiendo los datos de entrada de un fichero concreto. Hay varios operadores para redireccionar la entrada y las salidas de un programa de distintas maneras:

- >: redirecciona `stdout` a un fichero; si el fichero existe, lo sobrescribe:

```
$ who > usuarios.txt; less usuarios.txt
```

- >>: redirecciona `stdout` a un fichero; si el fichero existe, añade los datos al final del mismo.
- 2 >: redirecciona `stderr` a un fichero; si el fichero existe, lo sobrescribe:

```
$ find / -type d -exec cat {} \; 2>errores.txt
```

- 2 >> : similar a >> pero para la salida de error.
- `n`>&`m`: redirecciona el descriptor de fichero `n` al descriptor de fichero `m`; en caso de que `n` se omita, se sobrentiende un 1 (`stdout`):

```
$ cat file directorio > salida.txt 2>&1
# Redirecciona stdout al fichero salida.txt y stderr a stdout
```

- <: lee la entrada estándar de un fichero:

```
$ grep cadena < fichero.txt # busca "cadena" dentro de fichero.txt
```

- |: redirecciona la salida estándar de una orden a la entrada estándar de la orden que le sigue:

```
$ who | grep pilar
```

Éste último tipo de redirección es quizás el más importante, puesto que se usa para integrar diferentes órdenes y programas, mediante la interconexión de sus entradas y salidas estándares. Más concretamente, con una tubería o `pipe` (símbolo |) hay varias órdenes que se ejecutan sucesivamente, de manera que la salida estándar de la primera se envía a la entrada estándar de la siguiente, y así hasta que se ejecuta la última:

```
$ orden1 | orden 2 | ... | orden n
```

En la siguiente sección veremos una serie de órdenes cuyo principal cometido es procesar el texto que les llega por la entrada estándar y volcar el resultado de dicho procesamiento en la salida estándar, en la forma de texto filtrado. Estas órdenes, por tanto, se prestan al uso intensivo del mecanismo de redirección a través de tuberías.

2. Órdenes internas de bash

Una orden interna del shell es una orden que el intérprete implementa y que ejecuta sin llamar a programas externos. Por ejemplo, `echo` es una orden interna de bash y cuando se llama desde un script no se ejecuta el fichero `/bin/echo`. Algunos de las órdenes internas más utilizadas son:

- `echo`: envía una cadena a la salida estándar, normalmente la consola o una tubería. Por ejemplo:

```
echo El valor de la variable es $auxvar
```

- `read`: lee una cadena de la entrada estándar y la asigna a una variable, permitiendo obtener entrada de datos por teclado en la ejecución de un guión shell:

```
echo -n "Introduzca un valor para var1: "
read var1
echo "var1 = $var1"
```

  La orden `read` puede leer varias variables a la vez. También se puede combinar el uso de `read` con `echo`, para mostrar un prompt que indique qué es lo que se está pidiendo. Hay una serie de caracteres especiales para usar en `echo` y que permiten posicionar el cursor en un sitio determinado:

  - \b: retrocede una posición (sin borrar).
  - \f: alimentación de página.
  - \n: salto de línea.
  - \t: tabulador.

  Para que `echo` reconozca estos caracteres es necesario utilizar la opción "-e" y utilizar comillas dobles:

```
$ echo -e "Hola \t  ¿cómo estás?"
Hola     ¿cómo estás?
```

  Una orden alternativa a `echo` para imprimir en la salida estándar es la orden `printf`. Su potencial ventaja radica en la facilidad para formatear los datos de salida al estilo del `printf` del lenguaje `C`. Por ejemplo, la orden:

```
printf "Número: \t%05d\nCadena: \t%s\n" 12 Mensaje
```

produciría una salida como la siguiente:

```
Número:         00012 Cadena:         Mensaje
```

- `cd`: cambia de directorio.
- `pwd`: devuelve el nombre del directorio actual, equivale a leer el valor de la variable $PWD.
- `pushd` / `popd` / `dirs`: estas órdenes son muy útiles cuando un script tiene que navegar por un árbol de directorios:
  - `pushd`: apila un directorio en la pila de directorios.
  - `popd`: lo desapila y cambia a ese directorio.
  - `dirs`: muestra el contenido de la pila.
- `let arg [arg]`: cada `arg` es una expresión aritmética a ser evaluada (véase el apartado 3):

  ```
  $ let a=$b+7
  ```

  Si el último `arg` se evalúa a 0, let devuelve 1; si no, devuelve 0.
- `test`: permite evaluar si una expresión es verdadera o falsa (véase el apartado "La orden `test`").
- `export`: hace que el valor de una variable esté disponible para todos los procesos hijos del shell.
- `[.|source] nombre_fichero argumentos`: lee y ejecuta órdenes desde `nombre_fichero` en el entorno actual del shell y devuelve el estado de salida de la última orden ejecutada desde `nombre_fichero`. Si se suministran argumentos, se convierten en los parámetros cuando se ejecuta `nombre_fichero`. Cuando se ejecuta un guión shell precediéndolo de "." o `source`, no se crea un shell hijo para ejecutarlo, por lo que cualquier modificación en las variables de entorno permanece al finalizar la ejecución, así como las nuevas variables creadas.
- `exit`: finaliza la ejecución del guión. Recibe como argumento un entero que será el valor de retorno. Este valor lo recogerá el proceso que ha llamado al guión shell.
- `fg`: reanuda la ejecución de un proceso parado, o bien devuelve un proceso que estaba ejecutándose en segundo plano al primer plano.
- `bg`: lleva a segundo plano un proceso de primer plano o bien un proceso suspendido.
- `wait`: detiene la ejecución hasta que los procesos que hay en segundo plano terminan.
- `true` y `false`: devuelven 0 y 1 siempre, respectivamente.

Nota: el valor 0 se corresponde con `true`, y cualquier valor distinto de 0 con `false`.

3. Evaluación aritmética

El shell permite que se evalúen expresiones aritméticas, bajo ciertas circunstancias. La evaluación se hace con enteros largos sin comprobación de desbordamiento, aunque la división por 0 se atrapa y se señala como un error. La tabla 1 siguiente muestra los operadores en orden de precedencia decreciente, agrupando operadores de igual precedencia. Por ejemplo, la siguiente orden muestra en pantalla el valor 64:

```
 $
echo $((2**6))
```

-, +	Menos y más unarios
~	Negación lógica y de bits
**	Exponenciación
*, /, %	Multiplicación, división, resto
+, -	Adición, sustracción
<<, >>	Desplazamientos de bits a izquierda y derecha
<=, >= , <, >	Comparación
==, !=	Igualdad y desigualdad
&	Y de bits (AND)
^	O exclusivo de bits (XOR)
|	O inclusivo de bits (OR)
&&	Y lógico (AND)
||	O lógico (OR)
`expre?expre:expre`	Evaluación condicional
=, +=, -=, *=, /=, %=, &=, ^=, |= <<=, >>=	Asignación: simple, después de la suma, de la resta, de la multiplicación, de la división, del resto, del AND bit a bit, del XOR bit a bit, del OR bit a bit, del desplazamiento a la izquierda bit a bit y del desplazamiento a la derecha bit a bit.

Cuadro 1: Operadores permitidos por bash, en orden de precedencia decreciente.

Se permite que las variables del shell actúen como operandos: se realiza la expansión de parámetros antes de la evaluación de la expresión. El valor de un parámetro se fuerza a un entero largo dentro de una expresión. Una variable no necesita tener activado su atributo de entero para emplearse en una expresión.

Las constantes con un `0` inicial se interpretan como números octales. Un `0x` ó `0X` inicial denota un número en hexadecimal. De otro modo, los números toman la forma [`base#`]`n`, donde `base` es un número en base 10 entre 2 y 64 que representa la base aritmética y `n` es un número en esa base. Si `base` se omite, entonces se emplea la base 10. Por ejemplo:

```
$ let a=6#10+1
$ echo el valor de a es $a
  El valor de a es 7
```

Los operadores se evalúan en orden de precedencia. Las subexpresiones entre paréntesis se evalúan primero y pueden sustituir a las reglas de precedencia anteriores.

Existen tres maneras de realizar operaciones aritméticas:

1. Con `let lista_expresiones`, como se ha dicho anteriormente, se pueden evaluar las expresiones aritméticas dadas como argumentos. Es interesante destacar que esta orden no es estándar, sino que es específica del `bash`. A continuación se muestra un ejemplo de su uso:

```
$ let a=6+7
$ echo El resultado de la suma es $a
  El resultado de la suma es: 13

$ let b=7%5
$ echo El resto de la división es: $b
El resto de la división es: 2

$ let c=2#101\|2#10
$ echo El valor de c es $c
El valor de c es 7
```

2. La orden `expr` sirve para evaluar expresiones aritméticas. Puede incluir los siguientes operadores: \(, \), *, \\, \+, \-, donde el carácter '\' se introduce para quitar el significado especial que pueda tener el carácter siguiente. Por ejemplo:

```
$ expr 10 \* \( 5 \+ 2 \)
70

$ i=`expr $i - 1`     #restará 1 a la variable i
```

   Dese cuenta que `expr` espera recibir cada número y operador como un argumento, por lo que no se pueden usar las comillas simples para eliminar el significado especial a todos los operadores de una expresión (aunque sí operador a operador).

3. Mediante `$((expresión))` también se pueden evaluar expresiones. Varios ejemplos de su uso serían:

```
$ echo El resultado de la suma es $((6+7))
  El resultado de la suma es: 13

$ echo El resto de la división es: $((7%5))
El resto de la división es: 2

$ echo El valor es $((2#101|2#10))
El valor de c es 7
```

4. La orden `test`

La orden `test` permite evaluar si una expresión es verdadera o falsa. Los tests no sólo operan sobre los valores de las variables, también permiten conocer, por ejemplo, las propiedades de un fichero.

Principalmente, se usan en la estructura `if/then/else/fi` para determinar qué parte del script se va a ejecutar. Un `if` puede evaluar, además de un `test`, otras expresiones, como una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética; básicamente cualquier orden que devuelva un código en `$?`.

La sintaxis de `test` puede ser una de las dos que se muestran a continuación:

```
test expresión
```

o bien:

```
[ expresión ]
```

IMPORTANTE: Los espacios en blanco entre la expresión y los corchetes son necesarios.
La expresión puede incluir operadores de comparación como los siguientes:

- Para **números**: `arg1 OP arg2`, donde `OP` puede ser uno de los siguientes:

`-eq`	Igual a
`-ne`	Distinto de
`-lt`	Menor que
`-le`	Menor o igual que
`-gt`	Mayor que
`-ge`	Mayor o igual que

  Es importante destacar que en las comparaciones con números, si utilizamos una variable que no está definida, saldrá un mensaje de error. El siguiente ejemplo, al no estar la variable `e` definida, mostrará un mensaje de error indicando que se ha encontrado un operador inesperado:

```
if [ $e -eq 1 ]
then
    echo Vale 1
else
    echo No vale 1
fi
```

Por el contrario, en el siguiente ejemplo, a la variable `e` se le asigna un valor si no está definida, por lo que sí funcionaría:

```
if [ ${e=0} -eq 1 ]
then
    echo Vale 1
else
    echo No vale 1
fi
```

- Para **caracteres alfabéticos** o **cadenas**:

`-z cadena`	Verdad si la longitud de `cadena` es cero.
`-n cadena`	Verdad si la longitud de `cadena` no es cero.
`cadena1 == cadena2`	Verdad si las cadenas son iguales. Se puede emplear = en vez de ==.
`cadena1 != cadena2`	Verdad si las cadenas no son iguales.
`cadena1 < cadena2`	Verdad si `cadena1` se ordena lexicográficamente antes de `cadena2` en la localización en curso.
`cadena1 > cadena2`	Verdad si `cadena1` se clasifica lexicográficamente después de `cadena2` en la localización en curso.

- En la expresión se pueden incluir **operaciones con ficheros**, entre otras:

`-e fichero`	El fichero existe.
`-r fichero`	El fichero existe y tengo permiso de lectura.
`-w fichero`	El fichero existe y tengo permiso de escritura.
`-x fichero`	El fichero existe y tengo permiso de ejecución.
`-f fichero`	El fichero existe y es regular.
`-s fichero`	El fichero existe y es de tamaño mayor a cero.
`-d fichero`	El fichero existe y es un directorio.

- Además, se pueden incluir **operadores lógicos** y **paréntesis**:

`-o`	OR
`-a`	AND
!	NOT
\(	Paréntesis izquierdo
\)	Paréntesis derecho

A continuación veremos distintos ejemplos de uso de la orden `test`, con el fin de aclarar su funcionamiento. Uno de los usos más comunes de la variable `$#`, es validar el número de argumentos necesarios en un programa shell. Por ejemplo:

```
if test $# -ne 2
then
     echo "se necesitan dos argumentos"
     exit
fi
```

El siguiente ejemplo comprueba el valor del primer parámetro posicional. Si es un fichero (`-f`) se visualiza su contenido; sino, entonces se comprueba si es un directorio y si es así cambia al directorio y muestra su contenido. En otro caso, `echo` muestra un mensaje de error.

```
if test -f "$1"       # ¿ es un fichero ?
then
     more $1
elif test -d "$1"     # ¿ es un directorio ?
then
     (cd "$1";ls -l|more)
else                  # no es ni fichero ni directorio
     echo "$1 no es fichero ni directorio"
fi
```

Comparando dos cadenas:

```
#! /bin/bash
S1='cadena'
S2='Cadena'
if [ $S1!=$S2 ];
then
       echo "S1('$S1') no es igual a S2('$S2')"
fi
if [ $S1=$S1 ];
then
       echo "S1('$S1') es igual a S1('$S1')"
fi
```

En determinadas versiones, esto no es buena idea, porque si `$S1` o `$S2` son vacíos, aparecerá un error sintáctico. En este caso, es mejor: `x$1=x$2` o ``` ``$1''=``$2'' ```.

5. Órdenes simples, listas de órdenes y órdenes compuestas

5.1. Órdenes simples

Una orden simple es una secuencia de asignaciones opcionales de variables seguida por palabras separadas por blancos y redirecciones, y terminadas por un operador de control. La primera palabra especifica la orden a ser ejecutada. Las palabras restantes se pasan como argumentos a la orden pedida. Por ejemplo, si tenemos un shell script llamado `programa`, podemos ejecutar la siguiente orden:

```
$ a=9 programa
```

La secuencia de ejecución que se sigue es: se asigna el valor a la variable `a`, que se exporta a `programa`. Esto es, `programa` va a utilizar la variable `a` con el valor 9.

El valor devuelto de una orden simple es su estado de salida ó 128+n si la orden ha terminado debido a la señal n.

5.2. Listas de órdenes

Un operador de control es uno de los siguientes símbolos:

&	&&	;	;;	(	)	|	<nueva-línea>

Una lista es una secuencia de una o más órdenes separadas por uno de los operadores `;`, `&`, `&&` o ||, y terminada opcionalmente por uno de `;`, `&`, o <nueva-línea>. De estos operadores de listas, `&&` y || tienen igual precedencia, seguidos por `;` y `&`, que tienen igual precedencia.

Si una orden se termina mediante el operador de control `&`, el shell ejecuta la orden en segundo plano en un subshell. El shell no espera a que la orden acabe y el estado devuelto es 0. Las órdenes separadas por un `;` se ejecutan secuencialmente; el shell espera que cada orden termine. El estado devuelto es el estado de salida de la última orden ejecutada.

Los operadores de control `&&` y || denotan listas Y (AND) y O (OR) respectivamente. Una lista Y tiene la forma:

```
orden1 && orden2
```

`orden2` se ejecuta si y sólo si `orden1` devuelve un estado de salida 0.

Una lista O tiene la forma:

```
orden1 || orden2
```

`orden2` se ejecuta si y sólo si `orden1` devuelve un estado de salida distinto de 0.

El estado de salida de las listas Y y O es el de la última orden ejecutada en la lista.

Dos ejemplos del funcionamiento de estas listas de órdenes son:

```
test -e prac.c || echo El fichero no existe
test -e prac.c && echo El fichero sí existe
```

La orden `test -e fichero` devuelve verdad si el fichero existe. Cuando no existe devuelve un 1 y la lista O tendría efecto, pero no la Y. Cuando el fichero existe, devuelve 0 y la lista Y tendría efecto, pero no la O.

Otros ejemplos de uso son:

```
sleep 1 || echo Hola    # echo no saca nada en pantalla
sleep 1 && echo Hola    # echo muestra en pantalla Hola
```

Un ejemplo de lista de órdenes encadenadas con ; es:

```
ls -l; cat prac.c; date
```

Primero ejecuta la orden `ls -l`, a continuación muestra en pantalla el fichero `prac.c` (`cat prac.c`) y por último muestra la fecha (`date`).

El siguiente ejemplo muestra el uso del operador de control & en una lista:

```
ls -l & cat prac.c & date &
```

En este caso, ejecuta las tres órdenes en segundo plano, que se ejecutan una tras otra sin esperar.

5.3. Órdenes compuestas

Las órdenes se pueden agrupar formando órdenes compuestas:

- `{ c1 ; ... ; cn; }`: las n órdenes se ejecutan simplemente en el entorno del shell en curso, sin crear un shell nuevo. Esto se conocen como una **orden de grupo**.
- `( c1 ; ... ; cn )`: las n órdenes se ejecutan en un nuevo shell hijo, se hace un `fork`.

¡OJO! Es necesario dejar los espacios en blanco entre las órdenes y las llaves o los paréntesis.

Para ver de forma clara la diferencia entre estas dos opciones lo mejor es estudiar qué sucede con el siguiente ejemplo:

Ejemplo	`#!/bin/bash`
	`cd /usr`
	`{ cd bin; ls; }`
	`pwd`
		`#!/bin/bash`
		`cd /usr`
		`( cd bin; ls )`
		`pwd`

Resultado	1.- Entrar al directorio `/usr` 2.- Listado del directorio `/usr/bin` 3.- Directorio actual: `/usr/bin`	1.- Entrar al directorio `/usr` 2.- Listado del directorio `/usr/bin` 3.- Directorio actual: `/usr`

Ambos grupos de órdenes se utilizan para procesar la salida de varios procesos como una sola. Por ejemplo:

```
$ ( echo bb; echo ca; echo aa; echo a ) | sort
  a
  aa
  bb
  ca
```

6. Funciones

Como en casi todo lenguaje de programación, se pueden utilizar funciones para agrupar trozos de código de una manera más lógica o practicar la recursión.

Declarar una función es sólo cuestión de escribir:

```
function mi_func
{ mi_código }
```

Llamar a la función es como llamar a otro programa, sólo hay que escribir su nombre.

6.1. Ejemplo de funciones

```
#! /bin/bash

# Se define la función salir
function salir {
       exit
}

# Se define la función hola
function hola {
       echo ¡Hola!
}

hola        # Se llama a la función hola
salir       # Se llama a la función salir
echo petete
```

Tenga en cuenta que una función no necesita ser declarada en un orden específico.

Cuando ejecute el script se dará cuenta de que: primero se llama a la función `hola`, luego a la función `salir` y el programa nunca llega a la línea `echo petete`.

6.2. Ejemplo de funciones con parámetros

```
#!/bin/bash
function salir {
       exit
}

function e {
       echo $1
}

e Hola
e Mundo
salir
echo petete
```

Este script es casi idéntico al anterior. La diferencia principal es la función `e`, que imprime el primer argumento que recibe. Los argumentos, dentro de las funciones, son tratados de la misma manera que los argumentos suministrados al script. (Véase el apartado "Variables y parámetros").

7. Depuración

Una buena idea para depurar los programas es la opción `-x` en la primera línea:

```
#!/bin/bash -x
```

Como consecuencia, durante la ejecución se va mostrando cada línea del guión después de sustituir las variables por su valor, pero antes de ejecutarla.

Otra posibilidad es utilizar la opción `-v` que muestra cada línea como aparece en el script (tal como está en el fichero), antes de ejecutarla:

```
#!/bin/bash -v
```

Otra opción es llamar al programa usando el ejecutable bash. Por ejemplo, si nuestro programa se llama `prac1`, se podría invocar como:

```
$ bash -x prac1
```

o bien:

```
$ bash -v prac1
```

Ambas opciones pueden ser utilizadas de forma conjunta, dentro del guión:

```
#!/bin/bash -xv
```

o bien, a posteriori, al invocar el guión shell con el ejecutable bash:

```
$ bash -xv prac1
```

8. Patrones de uso del shell

En esta sección se introducen patrones de código para la programación shell. ¿Qué es un patrón? Un patrón es una solución documentada para un problema típico. Normalmente, cuando se programa en shell se encuentran problemas que tienen una muy fácil solución, y a lo largo del tiempo la gente ha ido recopilando las mejores soluciones para ellos. Los patrones expuestos en este apartado han sido extraidos en su mayoría de `http://c2.com/cgi/wiki?UnixShellPatterns`, donde pueden encontrarse algunos otros adicionales con también bastante utilidad.

8.1. Comprobación de cadena vacía

La cadena vacía a veces da algún problema al tratar con ella. Por ejemplo, considérese el siguiente trozo de código:

```
if [ $a = "" ] ; then echo "cadena vacia" ; fi
```

¿Qué pasa si la variable `a` es vacía? pues que la orden se convierte en:

```
if [ = "" ] ; then echo "cadena vacia" ; fi
```

Lo cual no es sintácticamente correcto (falta un operador a la izquierda de "="). La solución es utilizar comillas dobles para rodear la variable:

```
if [ "$a" = "" ] ; then echo "cadena vacia" ; fi
```

o incluso mejor, utilizar la construcción:

```
if [ "x$a" = x ] ; then echo "cadena vacia" ; fi
```

La `x` inicial impide que el valor de la variable se pudiera tomar como una opción.

8.2. Uso de `xargs`

Muchas de las órdenes de UNIX aceptan varios ficheros. Por ejemplo, imaginemos que queremos listar todos los directorios que están especificados en una variable:

```
dirs="a b c"
for i in $dirs ; do
        ls $i
done
```

Esto tiene un problema: lanza tres (o n) subshells y ejecuta n veces `ls`. Esta orden también acepta la sintaxis:

```
dirs="a b c"
ls $dirs
```

Una alternativa a esto cuando, por ejemplo, los argumentos están en un fichero, es utilizar `xargs`. Imaginemos que el fichero `directorios` contiene los directorios a listar. El programa `xargs` acepta un conjunto de datos en la entrada estándar y ejecuta la orden con todos los parámetros añadidos a la misma:

```
cat directorios | xargs ls
```

Esto ejecuta el programa `ls` con cada línea del fichero como argumento.

8.3. Leer un fichero línea a línea

A veces surge la necesidad de leer y procesar un fichero línea a línea. La mayoría de las utilidades de UNIX tratan con el fichero como un todo, y aunque permiten separar un conjunto de líneas, no permiten actuar una a una. La orden `read` ya se vió para leer desde el teclado variables, pero gracias a la redirección se puede utilizar para leer un fichero. Éste es el patrón:

```
while read i ; do
        echo "Línea: $i"
        # Procesar $i (línea actual)
done < $fichero
```

El bucle termina cuando la función `read` llega al final del fichero de forma automática.

8.4. Comprobar si una determinada variable posee un valor numérico válido

Esto puede ser muy útil para comprobar la validez de un argumento numérico. Por ejemplo:

```
if echo $1 | grep -x -q "[0-9]\+"
then
        echo "El argumento $1 es realmente un número natural."
else
        echo "El argumento $1 no es un número natural correcto."
fi
```

8.5. Leer opciones de la línea de argumentos

Aunque puede hacerse mediante programación convencional, el `bash` ofrece una alternativa interesante para esta tarea. Se trata de la orden `getopts`. La mejor manera de ver cómo se utiliza es con un ejemplo:

```
while getopts t:r:m MYOPTION
do
case $MYOPTION in
  t) echo "El argumento para la opción -t es $OPTARG"
     ;;
  r) echo "El índice siguiente al argumento de -r es $OPTIND"
     ;;
  m) echo "El flag -m ha sido activado"
     ;;
  ?) echo "Lo siento, se ha intentado una opción no existente";
     exit 1;
     ;;
esac
done
```

Podemos ahora probar el efecto de una invocación del guión como ésta:

```
./guion -m -r hola -t adios -l
```

La salida sería la siguiente:

```
El flag -m ha sido activado
El índice siguiente al argumento de -r es 4
El argumento para la opción -t es adios
./guion: opción ilegal -- l
Lo siento, se ha intentado una opción no existente
```

(El mensaje de la cuarta línea es en realidad enviado a la salida estándar de error, por lo que si se quisiera se podría eliminar redireccionando con `2>/dev/null`). Como puede observarse, `getopts` comprueba si las opciones utilizadas están en la lista permitida o no, y si han sido llamadas con un argumento adicional (indicado por los `:`) en la cadena de entrada "`t:r:m`". Las variables `$MYOPTION`, `$OPTIND` y `$OPTARG` contienen en cada paso del bucle, respectivamente, el carácter con la opción reconocida, el lugar que ocupa el siguiente argumento a procesar y el parámetro correspondiente a la opción reconocida (si ésta iba seguida de `:` en la cadena de entrada).

9. Ejemplos de guiones shell

1. Programa que copia un fichero en otro, controlando que el número de argumentos sea exactamente dos.

```
if [ $# != 2 ]
then
    echo "utilice: copia [desde] [hasta]"
    exit 1
fi
desde=$1
hasta=$2
if [ -f "$hasta" ]
then
    echo "$hasta ya existe, ¿ desea sobreescribirlo (s/n)?"
    read respuesta
if [ "$respuesta" != "s" ]
then
     echo "$desde no copiado"
     exit 0
fi
fi
cp $desde $hasta
```

2. Programa que imprime en pantalla el contenido de un fichero de datos o el contenido de todos los ficheros de un directorio.

```
if test -f "$1"
then
       pr $1|less
elif test -d "$1"
then
       cd $1; pr *|less
else
       echo "$1 no es un fichero ni un directorio"
fi
```

3. Programa que borra con confirmación todos los ficheros indicados como argumentos en la línea de órdenes.

```
#!/bin/bash
while test "$1" != ""
do
    rm -i $1
    shift
done
```

4. Programa que hace múltiples copias de ficheros a pares. En cada iteración desaparecen el primer y segundo argumento.

```
while test "$2" != ""
do
    echo $1 $2
    cp $1 $2
    shift; shift
done
if test "$1" != ""
then
    echo "$0: el número de argumentos debe ser par y > 2"
fi
```

5. Escribir un guión shell llamado `ldir` que liste los directorios existentes en el directorio actual.

```
#!/bin/bash
for archivo in *
do
      test -d $archivo && ls $archivo
done
```

6. Escribir un guión shell llamado `ver` que, para cada argumento que reciba, realice una de las siguientes operaciones:
   - si es un directorio ha de listar los ficheros que contiene,
   - si es un fichero regular lo tiene que mostrar por pantalla,
   - en otro caso, que indique que no es ni un fichero ni un directorio.

```
#!/bin/bash
for fich in $*
do
     if [ -d $fich ]
     then
         echo  "usando ls"
         ls $fich
     elif [ -f $fich ]
     then
         cat $fich
     else
         echo $fich no es ni un fichero ni un directorio
     fi
done
```

7. Escribir un guión shell que solicite confirmación si va a sobrescribir un fichero cuando se use la orden `cp`.

```
#!/bin/bash
if [ -f $2 ]
then
       echo "$2 existe. ¿Quieres sobreescribirlo? (s/n)"
       read sn
       if [ $sn = "N" -o $sn = "n" ]
       then
              exit 0
       fi
fi
cp $1 $2
```

8. Hacer un programa que ponga el atributo de ejecutable a los archivos pasados como argumento.

```
for fich in $@
do
     if test -f $fich
     then
          chmod u+x $fich
     fi
done
```

10. Ejercicios propuestos

1. Cree un shell script llamado `num_arg`, que devuelva el número de argumentos con el que ha sido llamado. Devolverá 0 (éxito) si se ha pasado algún argumento y 1 (error) en caso contrario. Mejorar este shell de forma que muestre una lista de todos los argumentos pasados o bien que indique que no tiene argumentos:

```
Los argumentos pasados son:
ARGUMENTO NÚMERO 1: X1
...
ARGUMENTO NÚMERO N: XN
  ó
No se han pasado argumentos
```

2. Cree un shell script llamado `doble` que pida un número por teclado y calcule su doble. Debe comprobar el número introducido y antes de terminar preguntará si deseamos calcular otro doble, en cuyo caso no terminará. Ejemplo:

```
Introduzca un número para calcular el doble: 89
El doble de 89 es 178
¿Desea calcular otro doble (S/N)?
```

3. Cree un shell script llamado `instalar` al que se le pasarán dos argumentos: fichero y directorio. El shell script debe copiar el fichero al directorio indicado. Además, debe modificar sus permisos de ejecución de forma que esté permitida al dueño y al grupo del fichero y prohibida al resto de usuarios. Antes de hacer la copia debe verificar los argumentos pasados, si se tiene permiso para hacer la copia, si el fichero es de texto o ejecutable, etc. Devolverá 0 (éxito) si todo ha ido bien y 1 (error) en caso contrario.

4. Cree un guión shell llamado `infouser` que reciba un único parámetro (el login de un usuario) y que muestre la siguiente información:

   - Login.
   - Nombre completo del usuario.
   - Directorio home.
   - Shell que utiliza.
   - Una línea que indique si el usuario está actualmente conectado o no.
   - Procesos pertenecientes a dicho usuario. La información a mostrar para cada proceso debe ser el PID y la línea de órdenes que dio lugar a la creación de dicho proceso.

   El guión debe comprobar:

   - Si las opciones y parámetros son correctos.
   - Si el usuario que se pasa como parámetro existe o no.

   Además, debe permitir las siguientes opciones:

   - `-p`: sólo muestra información de procesos.
   - `-u`: muestra toda la información excepto la referente a los procesos.
   - `--help`: muestra información de ayuda (lo que hace el guión, su sintaxis y significado de opciones y parámetros).

   Los códigos de retorno deben ser:

   - 0: éxito.

- 1: no se ha respetado la sintaxis de la orden.
- 2: usuario no existe.

Nota: parte de la información del usuario se puede obtener del fichero `/etc/passwd`. Pueden ser de utilidad las órdenes `getopts` y `finger`.

5. Cree un shell script llamado `bustr`, al que se le pase como parámetro una cadena y una lista de 0 a n nombres de fichero. El shell script debe devolvernos los nombres de los archivos que contienen en su interior la cadena especificada. Para evitar errores, sólo se hará con los archivos que sean regulares y sobre los que tengamos permiso de lectura. Por ejemplo:

```
bustr cadena fichero1 fichero2 fichero3
```

devolvería:

```
La cadena "cadena" se ha encontrado en los siguientes ficheros:
       fichero2
       fichero3
```

6. Construir un guión shell en Linux con la siguiente sintaxis

```
diffd [-i] directorio1 [directorio2]
```

- Función: debe mirar todos los nombres de fichero contenidos en el `directorio1` y en el `directorio2` (a excepción de los nombres de directorios) y mostrar las diferencias, es decir, mostrar el nombre de aquellos ficheros que aparecen en uno de los directorios pero no en el otro, indicando para cada uno de ellos el directorio en el que se encuentra.
- Parámetros: `directorio1` y `directorio2`: directorios entre los que se hace la comparación. Si se omite `directorio2` (que es opcional) se entenderá que queremos comparar `directorio1` con el directorio en el que nos encontremos al ejecutar `diffd`.
- Opciones: `-i`: invierte el funcionamiento de `diffd` haciendo que muestre los nombres de aquellos ficheros que se encuentran en los dos directorios (es obvio que en este caso no hay que indicar el directorio en el que aparece el fichero, pues aparece en ambos).

7. Hacer un guión shell llamado `mirm` que mueva al directorio `~/.borrados` los ficheros (nunca directorios) que se le indiquen como parámetros. Este guión shell viene acompañado por otro guión llamado `limpiezad` que se debe ejecutar en segundo plano y que cada 10 segundos compruebe si tiene que borrar ficheros del directorio `~/.borrados` en función del espacio ocupado en disco. Ambos guiones shell hacen uso de la variable de entorno `MINLIBRE` que indica el mínimo espacio en disco que debe quedar. De tal manera que una condición que deben cumplir ambos guiones es que, tras su ejecución, el espacio libre en disco (en bloques de 1K) debe ser igual o superior a `MINLIBRE`. En el caso de que, para cumplir dicha condición, haya que borrar ficheros de `~/.borrados`, se seguirá una política FIFO.

   Nota: basta con que ambos guiones funcionen para la partición en la que se encuentra el directorio personal del usuario.

   Además, ambos guiones deben hacer uso del fichero `~/.listaborrados` que guardará, sólo para cada fichero presente en `~/.borrados`, su ruta de acceso original.

   Mejora: debe permitirse el borrado de ficheros con igual nombre.

8. Como complemento del ejercicio anterior, hacer un guión shell llamado `recuperar` que se utilizará para recuperar ficheros borrados. Si se le pasa un único parámetro se entenderá que se trata de un patrón y entonces mostrará todos los ficheros de `~/.borrados` que cumplan dicho

patrón. Si se le pasan dos parámetros, el primero se entenderá como antes pero el segundo debe ser el nombre de un directorio y, en este caso, se recuperarán todos los ficheros borrados que cumplan el patrón y se dejarán en el directorio indicado como segundo parámetro.

Además, este guión debe implementar también la opción `-o <patrón>` que recuperará todos los ficheros que cumplan `<patrón>` y los copiará a su posición original, según lo indicado por el fichero `~/.listaborrados`. Si el directorio original ya no existe, debe crearse siempre que se tenga permiso.

Mejora: debe permitirse la recuperación de ficheros con igual nombre.

11. Bibliografía

- Página de manual del intérprete de órdenes bash (`man bash`).
- *Unix shell patterns*, J. Coplien *et alt.*
- *Programación en BASH - COMO de introducción*, Mike G. (traducido por Gabriel Rodríguez).
- *Shell scripting*, H. Whittal.
- *El libro de UNIX*, S. M. Sarwar *et alt.*