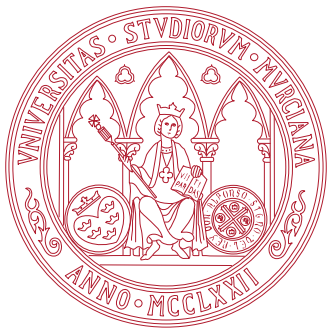

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

Prácticas de

Introducción a los Sistemas Operativos

2º de Grado en Ingeniería Informática

Boletín de prácticas 4 – Programación avanzada de *shell scripts* en Linux

Curso 2021/2022

Índice

1. Introducción

Una vez examinados los aspectos más básicos sobre el desarrollo de guiones shell con Bash en el boletín 3, veremos ahora ciertas características avanzadas, como los tipos de variables en Bash, la gramática que define y los mecanismos de depuración que pone a disposición del desarrollador para encontrar posibles fallos en sus *scripts*. Finalizaremos el boletín presentando diversos patrones de código para la programación con Bash, a través de los cuales se pueden resolver escenarios comúnmente encontrados durante el desarrollo de guiones shell.

1.1. Objetivos

Al terminar el boletín el alumno debe ser capaz de:

- Diferenciar las variables locales de las de entorno, y manejar ambos tipos de variables.
- Entender los distintos elementos que define la gramática de Bash (órdenes simples, tuberías, listas de órdenes y órdenes compuestas).
- Usar apropiadamente los mecanismos de depuración de Bash.
- Resolver determinadas situaciones en el desarrollo de guiones shell aplicando uno o varios de los patrones de uso del shell.

1.2. Órdenes utilizadas

Las órdenes que veremos en este boletín son:

- `export`
- `env`
- `read`
- `false`
- `mktemp`
- `set`
- `echo`
- `true`
- `bash`

En las páginas de manual de cada una de estas órdenes encontrarás información detallada de cómo usarlas.

2. Variables. Uso avanzado

Las variables que podemos manejar en Bash se dividen en dos tipos:

- **Variables locales**: están almacenadas en la zona de memoria local del proceso shell y son heredadas por los procesos hijos que va creando. Sus valores, sin embargo, desaparecen en el momento en que estos realizan una llamada al sistema `exec`.
- **Variables de entorno**: están almacenadas en el bloque de entorno del proceso shell (una estructura dentro del núcleo de Linux) y también son heredadas por sus procesos hijos. A diferencia de las anteriores, estas mantienen sus valores incluso después de que los procesos hijos ejecuten una llamada al sistema `exec`.

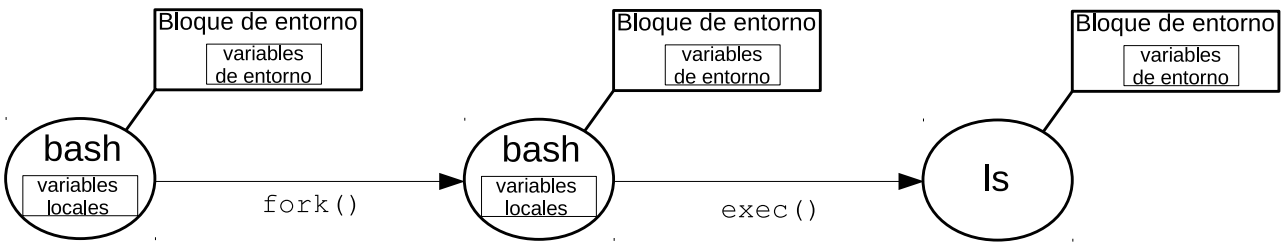

Figura 1: Variables locales y variables de entorno.

La figura 1 ilustra la diferencia entre las variables locales y las variables de entorno. Una vez que el proceso `bash` hijo realiza la llamada al sistema `exec` para sustituir el código por el de la orden `ls`, las variables locales se pierden mientras que las de entorno persisten.

Mediante la orden `export` podemos declarar nuevas variables de entorno o convertir una variable local en variable de entorno (en este último caso la variable pasará de estar en la zona de memoria local del proceso a residir dentro del bloque de entorno del mismo). Algunos ejemplos:

```
$ msg=123456        # Crea la variable local msg con
                    # el valor "123456"
$ bash              # Creamos un proceso Bash hijo
$ echo $msg         # No muestra nada dado que msg
                    # es una variable local del padre
$ exit              # Volvemos al shell padre
$ echo $msg         # Muestra el valor de la variable
123456              # local
$ export msg        # Convierte la variable msg en variable de entorno
$ bash              # Creamos un proceso Bash hijo
$ echo $msg         # Muestra el valor de la variable de
123456              # entorno msg
$ export var=valor # Crea la variable var, le asigna "valor"
                    # y la convierte a variable de entorno
$ exit              # Volvemos al shell padre
$ echo $var         # No muestra nada. El proceso padre no
                    # «ve» variables de entorno del hijo
```

La orden `set` muestra todas las variables (locales y de entorno) mientras que la orden `env` muestra sólo las variables de entorno.

Además de las variables que puede definir el programador, un shell tiene definidas, por defecto, una gran número de variables, muchas de ellas de entorno. Algunas de las variables de entorno más importantes son:

- `HOME`: directorio de trabajo del usuario actual que la orden `cd` toma por defecto.
- `LANG`: determina el idioma utilizado por los programas para comunicarse con el usuario. Así, por ejemplo:

  ```
  $ echo $LANG       # Leemos el idioma utilizado
  es_ES.UTF-8
  $ LANG=fr_FR.UTF-8 # Modificamos el valor de LANG
                     # para usar el francés como idioma
  $ lj               # Orden inexistente: mensaje de error en francés
  bash: lj: commande inconnue...
  ```

- `LOGNAME`: nombre del usuario.
- `PWD`: directorio actual.
- `PATH`: contiene un conjunto de rutas de directorios, separadas por ":". Cuando el usuario invoca una orden o programa a ejecutar indicando solo su nombre (sin ruta ni absoluta, ni relativa), el shell buscará el fichero ejecutable correspondiente en este conjunto de rutas, inspeccionándolas en orden secuencial.

  Así, si por ejemplo tenemos:

  ```
  $ echo $PATH
  /usr/bin:/usr/sbin:/home/alumno/misejecutables
  ```

y ejecutamos:

```
$ programilla
```

el shell buscará el ejecutable `programilla` en primer lugar en `/usr/bin`. Si lo encuentra ahí, lo ejecuta. En caso contrario, lo buscará en `/usr/sbin` de igual forma y, finalmente, si sigue sin encontrarlo, lo buscará en `/home/alumno/misejecutables`. Si aún así no se encuentra, se nos mostrará el mensaje de error correspondiente.

Obviamente, también podemos ejecutar cualquier orden o programa (siempre y cuando tengamos los permisos necesarios) si especificamos su ruta absoluta o relativa, aunque este se encuentre en un directorio no incluido en la variable `PATH`.

3. 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, por lo que cuando es llamada desde un script no se ejecuta el fichero `/bin/echo`. Realmente, muchas de estas órdenes también son externas (tienen su ejecutable en `/bin/`), pero sus códigos han sido incorporados al del `/bin/bash` para acelerar sus ejecuciones. De esta forma, por ejemplo, cuando invocamos la orden `echo` directamente, se ejecuta el código de esta que tiene incorporado en `/bin/bash`, mientras que si ejecutamos `/bin/echo` estaremos llamando al ejecutable externo.

Algunas de las órdenes internas más utilizadas son:

- `echo`: envía una o varias cadenas de caracteres que recibe como parámetros a la salida estándar, normalmente la consola o una tubería. Su sintaxis es:

  ```
  echo [opción]... [cadena]...
  ```

  Las opciones más comunes son:

  - `-n`: no se realiza el salto de línea automático tras mostrar el texto por pantalla.
  - `-e`: habilita la interpretación de una serie de secuencias especiales de dos caracteres que permiten posicionar el cursor en un lugar determinado:
    - `\b`: retrocede una posición (sin borrar).
    - `\f`: alimentación de página.
    - `\n`: salto de línea.
    - `\t`: tabulador.

    Para que `echo` reconozca estas secuencias de dos caracteres es necesario, que además de utilizar la opción `-e`, se encierren la(s) cadena(s) argumento entre 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
  ```

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

```
echo -n "Introduzca su nombre:"
read nombre
echo "Hola $nombre !"
```

- `cd`: cambia de directorio, tal como se vio en el primer boletín.
- `pwd`: devuelve el nombre del directorio actual. Equivale a leer el valor de la variable `PWD`.
- `let arg [arg ...]`: cada `arg` es una expresión aritmética a ser evaluada, tal como se vio en el boletín anterior.
- `test`: permite evaluar si una expresión es verdadera o falsa, tal como se vio en el boletín anterior.
- `export`: hace que el valor de una variable esté disponible para todos los procesos hijos del shell, tal como se vio en la sección anterior.
- `exit`: finaliza la ejecución del guion, tal como se vio en el boletín anterior.
- `true` y `false`: devuelven como código de retorno 0 y 1, respectivamente.

  Nota: En general, en Linux, el valor 0 se corresponde con `true`, y cualquier valor distinto de 0 con `false`, tal como vimos en el boletín anterior acerca del valor devuelto tras la ejecución de una orden del shell (recogido en la variable `$?`), donde un 0 indica que la orden se ha ejecutado con éxito y otro valor indica que ha habido errores.

4. Gramática de Bash

Para lograr comprender con detalle cómo Bash lleva a cabo la ejecución de nuestros scripts, hemos de prestar atención a la gramática definida por este, y que está conformada por los siguientes elementos:

4.1. Órdenes simples

Podemos decir que una *orden simple* está formada por cuatro elementos, que han de especificarse exactamente en el orden en el que se describen. El primer elemento, que es opcional, es una secuencia de asignaciones a variables. El segundo elemento es una lista de palabras separadas por espacios en blanco, donde la primera palabra es la orden a ejecutar y el resto son los argumentos de la misma. El tercer elemento, también opcional, lo conforman posibles redirecciones. Finalmente, el cuarto elemento es un operador de control.

En cuanto al primer elemento, aquellas variables que no existan, se crearán, y aquellas que no sean de entorno, se exportarán. Eso sí, los valores de estas variables solo serán visibles para la orden que aparece en el segundo elemento. En cuanto al último elemento, los operadores de control vienen definidos por los siguientes símbolos:

`||`	`&`	`&&`	`;`	`;;`	`(`	`)`	`|`	`|&`	<nueva-línea>

Por ejemplo, a través de la siguiente orden simple abrimos el fichero `referencia-ordenes.pdf` del directorio actual con el programa `evince`, haciendo que este se configure en idioma francés (se asigna el valor a la variable de entorno `LANG`, que para el nuevo proceso creado tendrá el valor `fr_FR.UTF-8`), su salida de error se redirija al dispositivo nulo y se ejecute en segundo plano (si una orden se termina mediante el operador de control `&`, el shell ejecuta la orden en segundo plano en un proceso hijo):

```
$ LANG=fr_FR.UTF-8 evince ./referencia-ordenes.pdf 2>/dev/null &
```

Tal como vimos en el boletín anterior, el valor devuelto de una orden simple es su estado de salida. En el caso de las órdenes ejecutadas en segundo plano, el shell no espera a que la orden acabe y el estado devuelto es 0.

4.2. Tuberías

Tal como se explicó en el boletín 2, una tubería es una secuencia de una o más órdenes (órdenes simples como acabamos de ver u órdenes compuestas como veremos a continuación) separadas por un operador de control | o |&. Por defecto, el estado de salida de una tubería es el estado de salida de la última orden de la misma.

4.3. Listas de órdenes

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

Las órdenes separadas por el operador ; se ejecutan secuencialmente: el shell espera a que cada orden termine antes de ejecutar la siguiente. El estado devuelto es el estado de salida de la última orden ejecutada. Un ejemplo de lista de órdenes encadenadas con el operador ; 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`).

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, es decir, si tiene éxito en su ejecución.

Una lista *O* tiene la forma:

```
orden1 || orden2
```

`orden2` se ejecuta si y sólo si `orden1` devuelve un estado de salida distinto de 0, o sea, si no tiene éxito en su ejecución.

El estado de salida de las listas *Y* y *O* es el de la última orden de la lista que se ha ejecutado (tal como hemos visto, la última orden ejecutada de una lista puede coincidir o no con la última orden que aparece en dicha lista).

Dos ejemplos del funcionamiento de estas listas de órdenes son:

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

La orden `test -f fichero` devuelve verdad si el fichero regular `prac.c` 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
```

Por último, 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, en paralelo, sin que una orden espere a otra. El código devuelto es siempre 0.

4.4. Órdenes compuestas

Las órdenes se pueden agrupar formando órdenes compuestas de la siguiente forma:

`( orden_1 ; ... ; orden_n )`: la lista de `n` órdenes se ejecuta en un nuevo shell en un proceso hijo, o sea, tras un `fork`. El estado de salida es el de la lista, que como hemos visto es el estado de salida de la última orden ejecutada.

Podemos emplear una orden compuesta 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
```

En Bash, `while`, `for`, `if` o `case` son órdenes compuestas, y por lo tanto, todas las órdenes que aparecen en cada una de ellas comparten la entrada estándar, la salida estándar y la salida estándar de error. A diferencia de las órdenes compuesta formadas por una lista de n órdenes que acabamos de ver, `while`, `for`, `if` o `case` pueden ser ejecutadas por el mismo shell sin necesidad de crear procesos hijos. El estado devuelto en este caso es el estado de salida de la última orden ejecutada. Si no se ejecuta ninguna orden (por ejemplo, no hay coincidencia con ninguno de los patrones especificados en una orden `case`) el estado devuelto es 0.

Por ejemplo:

```
#!/bin/bash

seq 1 1 10 | while read i
do
    echo $i
    read a
done
```

Vemos que por pantalla se imprimen los números `1`, `3`, `5`, `7` y `9`, pero no los pares. Esto es debido a que las órdenes `read i` y `read a` obtienen sus valores de la misma entrada estándar, que como podemos observar está conectada por la tubería con la salida estándar de la orden precedente, por lo que los números pares se van asignando a la variable `a` en cada iteración del bucle.

Recuerda

Si utilizamos, por ejemplo, `while` después de una tubería como en el ejemplo que acabamos de ver, se creará un proceso hijo para su ejecución, aunque observa que en este caso el proceso hijo se crea como consecuencia del empleo de la tubería, no por el hecho de que `while` sea una orden compuesta.

5. Depuración

En el proceso de realización de un guion shell es bastante habitual que veamos que este no se comporta como esperamos, bien porque falle bien porque los resultados del mismo no sean los correctos. En estos casos debemos recurrir a la depuración del guion utilizando opciones específicas que Bash pone a nuestra disposición, concretamente `-u`, `-x` y `-v`. Estas opciones de Bash se pueden usar de forma individual o combinadas.

La opción `-u` es francamente útil y nos puede ayudar a detectar errores en el código que fácilmente podrían pasar inadvertidos tras una lectura del mismo. En concreto, esta opción sirve para detectar

variables o parámetros que son utilizados *sin estar inicializados previamente*. Con frecuencia esta situación es síntoma de un nombre de variable mal escrito. Así por ejemplo, en este guion shell, llamado `prueba_u`, que parece estar correcto a priori:

```
#!/bin/bash
operador1=100
operador2=200
let resultado=operado1+operador2
echo "El resultado de sumar $operador1 y $operador2 es $resultado"
```

la ejecución nos mostraría algo extraño:

```
$ prueba_u
El resultado de sumar 100 y 200 es 200
```

pero, si usamos la opción `-u` podríamos detectar fácilmente el error:

```
$ bash -u prueba_u
prueba_u: line 4: operado1: unbound variable
```

Es decir, en la suma realizada con la instrucción `let` estamos usando erróneamente la variable no definida `operado1`, en lugar de `operador1`. Cuando en una operación aritmética aparece una variable no definida, esta se toma con valor 0.

Alternativamente, a través de la opción `-x` podríamos depurar nuestro programa línea a línea. En este caso Bash va mostrando, durante la ejecución del guion, cada una de las líneas después de realizar las correspondientes sustituciones (por ejemplo, las variables por su valor), pero antes de ejecutarlas.

La última opción, `-v`, es similar a la anterior, pero en este caso muestra cada línea como aparece en el script (tal como está en el fichero), antes de ejecutarla.

Como decíamos anteriormente, estas opciones se pueden combinar:

```
$ bash -ux prac1
```

En caso de que se vaya a dar permiso de ejecución al guion, y por tanto se ejecute directamente sin anteponer `bash`, podemos indicar las opciones de depuración en la primera línea del guion:

```
#!/bin/bash -u
```

Recuerda

Si ejecutamos un guion anteponiendo la orden `bash`, la primera línea del mismo (`#!/bin/bash`) se ignorará tal y como se explicó en el boletín 2. De esta manera las opciones de depuración que hubiésemos incluido en esta primera línea no tendrían efecto, y caso de necesitarlas, deberemos indicarlas como opciones de la orden `bash` como acabamos de ver.

6. 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 fácil solución, y a lo largo del tiempo la gente ha ido recopilando las mejores soluciones para ellos.

6.1. 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 test $a = "" ; then echo "cadena vacia" ; fi
```

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

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

6.2. Recorrer la salida de una orden

En el código de un guion shell es frecuente tener que procesar línea a línea el resultado de la ejecución de una orden o conjunto de órdenes. Esto podemos hacerlo fácilmente como se ilustra con el siguiente guion shell (`procesa_clientes.sh`):

```
#!/bin/bash
numerocliente=0
cat $1 | while read cliente
do
    let numerocliente=$numerocliente+1
    echo "#$numerocliente : $cliente"
done
```

su ejecución produciría la siguiente salida:

```
$ cat clientes.txt
Pepe Pérez
Juan López
Jesús Sánchez
Alfredo Gómez
$ bash procesa_clientes.sh clientes.txt
#1 : Pepe Pérez
#2 : Juan López
#3 : Jesús Sánchez
#4 : Alfredo Gómez
```

Cuidado: Uso de una variable para almacenar la salida de una orden.

Alternativamente, se podría haber pensado en almacenar la salida de la orden a procesar (`cat` en el ejemplo) en una variable y, posteriormente, realizar los tratamientos oportunos del contenido de esta variable, tal como podemos ver a continuación:

```
...
clientes=$(cat $1)
echo "$clientes" | while read cliente
...
```

Es preciso sin embargo tener en cuenta que las variables tienen una capacidad máxima de almacenamiento, con lo que si el fichero fuese demasiado extenso y su contenido no cupiese en la

. variable, el resultado no sería el esperado.

Supongamos que ampliamos el guion de ejemplo para que nos informe, además, acerca del número total de clientes de la siguiente forma:

```
#!/bin/bash
numerocliente=0
cat $1 | while read cliente
do
    let numerocliente=$numerocliente+1
    echo "#$numerocliente : $cliente"
done
echo "Número total de clientes: $numerocliente"
```

su ejecución producirá la siguiente salida:

```
$ bash procesa_clientes.sh clientes.txt
#1 : Pepe Pérez
#2 : Juan López
#3 : Jesús Sánchez
#4 : Alfredo Gómez
Número total de clientes: 0
```

Este error a la hora de mostrar el número total de clientes ha ocurrido porque las órdenes enlazadas mediante tuberías se ejecutan en sus respectivos procesos, todos ellos hijos del proceso principal que está interpretando el guion shell. Por lo que la actualización de la variable `numerocliente` dentro del segundo proceso de la tubería (el bucle `while`) afecta únicamente a la copia de esta variable en dicho proceso, pero no a la copia de esta variable en el proceso padre. Por tanto, en el proceso padre, la variable `numerocliente` se ha mantenido a 0 desde su inicialización, y eso es lo que se muestra en la última línea.

Para evitar esta problema podemos forzar a que el bucle `while` y la siguiente línea de código se ejecuten dentro del mismo proceso formando una orden compuesta:

```
#!/bin/bash
numerocliente=0
cat $1 | (while read cliente
do
    let numerocliente=$numerocliente+1
    echo "#$numerocliente : $cliente"
done;
echo "Número total de clientes: $numerocliente")
```

Este mismo patrón podemos aplicarlo al recorrido de directorios empleando la orden `find`. Por ejemplo, el siguiente guion shell (`tamano_total.sh`) calcularía el tamaño total de todos los ficheros regulares encontrados a partir de un directorio que se le pasa como parámetro, recorriendo de forma recursiva los subdirectorios[1]:

```
#!/bin/bash
tamanototal=0
find "$1" -type f -printf "%s\n" | (while read tamano ; do
    let tamanototal+=tamano
done;
echo "Tamaño total: $tamanototal bytes")
```

[1]Recuerda que con la opción `-maxdepth` de `find` podemos controlar la *profundidad* del recorrido (por ejemplo, para limitar el recorrido únicamente al directorio pasado como argumento bastaría con añadir `-maxdepth 1`.

Un ejemplo de ejecución de dicho guion sería el siguiente:

```
$ bash tamano_total.sh /etc/default/
Tamaño total: 6379 bytes
```

6.3. Leer un fichero línea a línea

El patrón que acabamos de ver podemos aplicarlo al procesamiento línea a línea de la salida de cualquier orden , y por lo tanto, usando la orden `cat` como vimos en el patrón anterior, podemos leer un fichero línea a línea. Sin embargo, cuando se trata de hacer esto tenemos otra posibilidad. En este caso podemos redirigir la entrada estándar de la orden compuesta `while` al contenido del fichero a recorrer de la siguiente forma:

```
#!/bin/bash
numerocliente=0
while read cliente
do
    let numerocliente=$numerocliente+1
    echo "#$numerocliente : $cliente"
done < $1
echo "Número total de clientes: $numerocliente"
```

El bucle termina cuando la función `read` llega al final del fichero de forma automática. Es importante destacar que con redireccionamiento la última orden `echo` muestra correctamente el total de clientes. Observa que, en este caso, tanto la orden compuesta `while` a través de la cual se calcula este resultado, como la orden `echo` que lo muestra por pantalla, se ejecutan en el contexto del mismo proceso.

6.4. Utilización de ficheros temporales

En la implementación de algunos guiones shell puede ser oportuno la utilización de algún fichero temporal donde almacenar resultados parciales para, a continuación, ir consultando el contenido de este fichero de cara a realizar una segunda parte del procesamiento previsto. Con el objetivo de asegurarnos de que no haya ningún problema de permisos a la hora de crear este fichero temporal, es conveniente utilizar la orden `mktemp` que, por defecto, crea este fichero temporal en el directorio `/tmp` y devuelve por la salida estándar el nombre del fichero creado.

Ejemplo: Supongamos que queremos escribir un guion llamado `cambiapalabra.sh` que cambie en el contenido de un fichero de texto, cuyo nombre se pasa como primer parámetro, cada aparición de una palabra (segundo parámetro) por otra palabra (tercer parámetro). Sintaxis:

```
cambiapalabra.sh fichero palabra_old palabra_new
```

Una implementación posible sería la siguiente:

```
#!/bin/bash

while read linea
do
    comienzo=""
    for palabra in $linea
    do
        if test "$palabra" == "$2"
        then
            echo -n "$comienzo$3"
        else
```

```
            echo -n "$comienzo$palabra"
        fi
        if test -z "$comienzo"
        then
            comienzo=" "
        fi
    done
    echo
done < "$1" > "convert-$1"
```

donde observamos cómo el guion va leyendo, línea a línea, el contenido del fichero pasado como primer parámetro (bucle `while`). Por cada línea del fichero realiza entonces una búsqueda de la palabra pasada como segundo parámetro, sustituyendo cada ocurrencia por la palabra pasada como tercer parámetro (bucle `for` interno). La variable `comienzo` nos sirve para marcar si una palabra debe ir precedida de espacio en blanco (no es la primera palabra de la línea) o no (la primera palabra de cada línea). Las líneas resultantes van siendo volcadas a través de la salida estándar de la orden compuesta `while` (mediante el uso de la orden `echo`). Al final del guion, se redirige la salida estándar de `while` al fichero con la salida, que tendrá como nombre el resultado de añadir el prefijo «`convert-`» al nombre del fichero original.

Imaginemos que nos piden que modifiquemos el guion para informar por pantalla acerca de las líneas en las que se han hecho cambios de palabra. En este caso tendríamos que volcar el contenido de las líneas resultado directamente en un fichero temporal, ya que la salida estándar de `while` debemos usarla para informar acerca de las líneas afectadas por los cambios. El guion quedaría de la siguiente forma:

```
#!/bin/bash

temp=$(mktemp)
nlinea=1
while read linea
do
    comienzo=""
    for palabra in $linea
    do
        if test "$palabra" == "$2"
        then
            echo -n "$comienzo$3" >> $temp
            echo "Línea $nlinea"
        else
            echo -n "$comienzo$palabra" >> $temp
        fi
        if test -z "$comienzo"
        then
            comienzo=" "
        fi
    done
    echo >> $temp
    let nlinea+=1
done < "$1" | uniq

mv $temp "convert-$1"
```

Observa cómo al comienzo del guion usamos la orden `mktemp` para crear el fichero temporal y cómo dentro del bucle vamos volcando las líneas con el resultado en dicho fichero temporal. Una vez

que hemos acabado el recorrido del fichero original es cuando el guion renombra el fichero temporal, dándole como nombre el resultado de añadir el prefijo «convert-» al nombre del fichero original (como anteriormente), y hace, por lo tanto, que deje de ser un fichero temporal.

6.5. 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, para números enteros positivos:

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

La orden `grep` realiza la búsqueda de un entero positivo en el texto que le llega por su entrada estándar a través de la tubería, procedente de la orden `echo`. Este texto corresponde al contenido del primer parámetro del guion, `$1`. Para ello:

- El patrón de búsqueda del `grep` consiste en la aparición de un dígito (conjunto posibles de valores `[0-9]`) una o más veces (operador de repetición '+').
- Con la opción `-x` se indica que, para cada línea de la entrada, la búsqueda será exitosa si toda la línea coincide con el patrón. Normalmente la salida de `echo $1` será una única línea, por lo que el primer parámetro debe cumplir exactamente el patrón de búsqueda.
- Con la opción `-q` se evita que la orden `grep` muestre el resultado del filtrado por pantalla (observa que nos interesa únicamente su código de retorno).

Para números enteros, en general, podemos ampliar la comprobación para el caso de que el número pueda llevar signo. Para ello, se indica en el patrón de búsqueda que delante de los dígitos puede haber un + o un −, repetidos 0 o 1 veces:

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

6.6. Tratamiento de errores

En aquellos guiones que precisan algún parámetro de entrada en la propia línea de invocación, es frecuente tener que hacer un chequeo de estos parámetros para comprobar que se ajustan en número y forma a lo que precisa el guion para su correcta ejecución. Este tratamiento de errores se suele implementar al principio del código del guion shell de manera que si los parámetros no son correctos se muestre el correspondiente mensaje de error en la salida estándar de error (descriptor número 2) y se aborte la ejecución, devolviendo un código de error mediante la orden `exit`. Como ya se vio anteriormente, el código de error devuelto por el guion podrá ser consultado en `$?`.

Ejemplo: Supongamos que queremos escribir un guion (`errores.sh`) que debe recibir como parámetros un número natural y el nombre de un fichero regular que ha de existir y se debe poder leer, la comprobación de errores en los parámetros al inicio del guion shell podría tener esta forma:

```
#!/bin/bash
if test $# -ne 2
then
    echo "ERROR: El guion requiere 2 parámetros." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 1
fi

if ! echo $1 | grep -x -q "[0-9]\+"
then
    echo "ERROR: El primer parámetro debe ser un número natural." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 2
fi

if test ! -f "$2" -o ! -r "$2"
then
    echo "ERROR: El segundo parámetro debe ser un fichero regular existente y
con permiso de lectura." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 3
fi

####################################
# resto del código del guion shell #
####################################
```

Y, por tanto, diferentes intentos de utilización de este guion producirían este efecto:

```
$ bash errores.sh
ERROR: El guion requiere 2 parámetros.
USO: errores.sh <numero_natural> <nombre_fichero>

$ echo $?
1

$ bash errores.sh fichero_no_existe 111
ERROR: El primer parámetro debe ser un número natural.
USO: errores.sh <numero_natural> <nombre_fichero>


$ echo $?
2

$ bash errores.sh 111 fichero_no_existe
ERROR: El segundo parámetro debe ser un fichero regular existente
y con permiso de lectura.
USO: errores.sh <numero_natural> <nombre_fichero>

$ echo $?
3

$ bash errores.sh 111 fichero_no_lectura
```

```
ERROR: El segundo parámetro debe ser un fichero regular existente
y con permiso de lectura.
USO: errores.sh <numero_natural> <nombre_fichero>

$ echo $?
3

$ bash errores.sh 111 errores.sh
$ echo $?
0
```

La comprobación de errores en el resto de código del guion se debe implementar de forma similar, de manera que si se produce alguna situación de error que conlleve que no se pueda continuar la ejecución con normalidad, se ponga en funcionamiento el protocolo de tratamiento de errores: mostrar el mensaje oportuno en la salida estándar de error y abortar la ejecución devolviendo el código de error establecido para cada caso.

Es importante también que nos aseguremos que nuestro guion shell devuelve el código 0 siempre que no se haya producido una situación de error. Para ello es conveniente que la última orden ejecutada en caso de una terminación normal sea `exit 0`. Su no inclusión podría hacer que el guion shell devolviese de manera inesperada un código distinto de 0 aun cuando no se hubiese encontrado error alguno durante la ejecución (recuerda que el código devuelto por un guion es el de la última orden ejecutada). Para ilustrar este escenario, supongamos que modificamos el guion `errores.sh` para implementar la última condición como una lista de órdenes (en lugar de como una orden compuesta con `if`):

```
#!/bin/bash
if test $# -ne 2
then
    echo "ERROR: El guion requiere 2 parámetros." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 1
fi

if ! echo $1 | grep -x -q "[0-9]\+"
then
    echo "ERROR: El primer parámetro debe ser un número natural." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 2
fi

test ! -f "$2" -o ! -r "$2" && echo "ERROR: El segundo parámetro debe
ser un fichero regular existente y con permiso de lectura." >&2 && echo "USO:
$0 <numero_natural> <nombre_fichero>" >&2 && exit 2
```

En este caso, aun cuando se invoque correctamente, el código devuelto sería 1:

```
$ bash errores.sh 111 errores.sh
$ echo $?
1
```

Ten en cuenta que en este caso la última orden ejecutada es `test`, y dado que no se cumple la condición (el fichero existe y se puede leer), devuelve ese código 1. Para evitar este tipo de situaciones, añadimos siempre la orden `exit 0` al final de nuestros guiones shell:

```
#!/bin/bash
if test $# -ne 2
then
    echo "ERROR: El guion requiere 2 parámetros." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 1
fi

if ! echo $1 | grep -x -q "[0-9]\+"
then
    echo "ERROR: El primer parámetro debe ser un número natural." >&2
    echo "USO: $0 <numero_natural> <nombre_fichero>" >&2
    exit 2
fi

test ! -f "$2" -o ! -r "$2" && echo "ERROR: El segundo parámetro debe
ser un fichero regular existente y con permiso de lectura." >&2 && echo "USO:
$0 <numero_natural> <nombre_fichero>" >&2 && exit 2

exit 0
```

7. Ejercicios

1. Escribe un guion shell que muestre por pantalla un listado de todos los ficheros regulares cuyo nombre acabe en «`.pdf`» a partir del directorio actual. Para cada fichero, además del nombre con la ruta completa, se mostrará su tamaño en bytes. El guion debe informar al final del tamaño total acumulado de todos los ficheros encontrados.

2. Escribe un guion shell que muestre por pantalla un número aleatorio con tantas cifras como se indique a través del único argumento que recibe. El guion debe comprobar que recibe exactamente un argumento y que este es un número natural. Si se pasa un número incorrecto de argumentos, el guion devolverá el código de error 1. Si se pasa un argumento pero no es un número natural, el código de error devuelto será el 2. Puedes usar la variable `$RANDOM` para generar números aleatorios.

3. Escribe un guion shell que reciba como argumento un directorio y que muestre un listado de los ficheros regulares que hay en el mismo. En dicho listado aparecerán los nombres de los ficheros ordenados alfabéticamente y separados por comas. El guion debe comprobar que recibe exactamente un argumento y que este es el nombre de un directorio existente y con permisos de lectura y ejecución. Si se pasa un número incorrecto de argumentos, el guion devolverá el código de error 1. Si se pasa un argumento pero no es un directorio existente o no dispone de los permisos adecuados, el código de error devuelto será el 2.

4. Escribe un guion shell llamado `mi_uniq` que reciba como parámetro un fichero de texto y que muestre el contenido del mismo por pantalla haciendo que las líneas repetidas consecutivas aparezcan una sola vez. El guion debe comprobar que recibe exactamente un argumento y que este es el nombre de un fichero regular existente y con permiso de lectura. Si se pasa un número incorrecto de argumentos, el guion devolverá el código de error 1. Si se pasa un argumento pero no es un fichero regular existente o que se pueda leer, el código de error devuelto será el 2. Evidentemente, no debes utilizar la orden `uniq`.

5. Escribe un guion shell llamado `a_mayuscula` que reciba como parámetros el nombre de un fichero de texto que debe existir, una palabra y el nombre de un fichero que no debe existir (es decir, no debe haber una entrada de directorio con este nombre). El guion producirá un fichero de texto al que se le dará el nombre pasado como tercer argumento y cuyo contenido será el del primer fichero de texto pero habiendo convertido a mayúsculas todas las instancias de la palabra pasada como segundo parámetro. El guion debe comprobar que recibe exactamente tres argumentos y que el primero de ellos es el nombre de un fichero regular existente y el último se corresponde con una entrada de directorio que no existe. Caso de error se devolverá el código de error 1.

6. Amplía el guion shell anterior para que informe por pantalla acerca del número de palabras cambiadas y el porcentaje que estas representan sobre el total de palabras del fichero.

7. Escribe un guion shell llamado `cambia_PDF_pdf_recursivo.sh`, que cambie el sufijo «`*.PDF`» por «`*.pdf`» a todos los ficheros que se encuentren en los directorios indicados como parámetros, así como en cualquiera de sus subdirectorios. Si se pasa como argumento el nombre del algún directorio no válido, el guion dará un mensaje de error y devolverá el código de error 1. Si no se pasa ningún argumento, el guion no hará nada. Además, el guion deberá eliminar cualquier mensaje de error que surja como consecuencia de la falta de permisos para explorar determinados directorios.

8. Escribe un guion shell llamado `ordena_por_extension` que muestre un listado con los ficheros regulares del directorio que recibe como argumento (sin descender en los subdirectorios que este pudiese tener) ordenados por extensión. Aquellos ficheros que no tengan extensión no serán listados, y aquellos que compartan extensión aparecerán ordenados por su nombre.

8. Bibliografía

- Página de manual del intérprete de órdenes Bash (`man bash`).
- *El libro de UNIX*, S. M. Sarwar *et alt*, ISBN: 8478290605. Addison-Wesley, 2005.
- *Linux: Domine la administración del sistema*, 2ª edición. Sébastien Rohaut. ISBN 9782746073425. Eni, 2012.
- *Unix shell patterns* (http://wiki.c2.com/?UnixShellPatterns), J. Coplien *et alt.*
- *Programación en BASH - COMO de introducción* (http://es.tldp.org/COMO-INSFLUG/COMOs/Bash-Prog-Intro-COMO/), Mike G. (traducido por Gabriel Rodríguez).
- *Shell & Utilities: Detailed Toc* (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html), The Open Group Base Specifications.
- *Espacio Linux. Portal y comunidad GNU* (http://www.espaciolinux.com/).
- *Linux Shell Scripting Tutorial (LSST) v2.0* (https://bash.cyberciti.biz/guide/Main_Page), Vivek Gite *et alt.*