

Taller de programación shell

3 de Abril de 2004

revisión 1

Asociación de Usuarios de Software Libre de Elche

<http://www.kleenux.org/>



Introducción: ¿Qué es un shell?

Es una parte fundamental de todo sistema operativo que se encarga de ejecutar órdenes básicas para el manejo del sistema.

Suelen incorporar características como:

- control de procesos
- redirección de ficheros
- lenguaje para escribir pequeños programas

Introducción: ¿Qué es un shell?

Ejemplos de shell:

command.com cmd.exe - DOS y derivados (aceptamos barco)

ksh - korn shell de UNIX

chs - C shell, similar en sintaxis al lenguaje de programación C

bsh - Bourne shell

tcsh, zsh, ash, ...

Vamos a utilizar:

bash - Bourne Again shell, el shell mayoritario de sistemas Linux

Manejo básico del shell

Cuatro comandos básicos para empezar:

- **echo** : repite los argumentos en la salida estándar (ej. pantalla)
- **ls** : lista el contenido de un directorio
- **cat** : muestra el contenido de un fichero
- **more** : muestra el contenido de un fichero haciendo pausas entre pantallas si el fichero es muy largo

La línea de comandos

IZQUIERDA

Se mueve a la izquierda (anda!)

DERECHA

Se mueve a la derecha

ARRIBA

Vamos hacia atrás en el historial de comandos

ABAJO

Volvemos hacia adelante en el historial de comandos

FIN

CTRL + e

Vamos al final de la línea

INICIO

CTRL + a

Vamos al principio de la línea

CTRL + d

Fin de fichero

CTRL + l

Borra la pantalla

Mayúsculas + RePag

Hace scroll hacia arriba en la pantalla

Mayúsculas + AvPag

Hace scroll hacia abajo en la pantalla

Patrones de sustitución

*

cualquier cadena de texto

Ejemplo:

```
$ echo /usr/*
```

?

un solo caracter cualquiera

Ejemplo:

```
$ echo /usr/?bin
```

```
$ echo /usr/????
```

[...]

cualquiera de los caracteres entre corchetes

Ejemplo:

```
$ echo /usr/[aeiou]*
```

Redirección de ficheros

¿Qué es un fichero?

En UNIX (Linux, *BSD) todo es un fichero

Ficheros estándar que están abiertos para todo programa en ejecución:

Entrada estándar /dev/stdin (ej. **Teclado**)

Salida estándar /dev/stdout (ej. **Pantalla**)

Salida de error /dev/stderr (ej. **Pantalla**)

¿Qué pasa cuando ejecutamos **ls**?

Redirección de ficheros

Podemos hacer que el shell sustituya los ficheros estándar por otros

Algunos tipos de redirecciones:

Redirección **stdout** a fichero:

```
$ ls > salida
```

Redirección **stderr** a fichero:

```
$ ls pirulotropical 2> error
```

Redirección **stdin** a fichero:

```
$ cat < /dev/stdin
```

Añadir redirección a fichero:

```
$ echo añadir datos >> salida
```

```
$ ls uh 2>> error
```

Redirección de ficheros

Redirección "documentos empotrados":

```
$ cat << FIN
```

Esto es un ejemplo de un gran documento empotrado o embebido.

Todo lo que se encuentre el shell hasta “la cadena de cierre” será pasado al comando en cuestión (en este caso **cat**) por su entrada estándar (stdin).

¡Hasta los saltos de línea!

```
FIN
```

Tuberías

¿Qué es una tubería?

Es un fichero especial con dos extremos de forma que lo que escribimos en un lado de la tubería va a parar al otro.

Ejemplo de tubería en el shell:

```
$ ls /usr/bin | more
```

La salida del comando `ls /usr/bin` va a parar a la entrada del comando `more`.

Esto permite combinar la funcionalidad de distintos comandos UNIX.

Control de procesos

¿Qué es un **proceso**?

Es un programa en ejecución con características propias (memoria, pila, puntero de programa, número identificador único, etc.).

```
[ shell ] - ejecuta ls -> [ ls : shell espera ] - fin ls -> [ shell ]
```

Desde el shell podemos, aparte de crear procesos, manipular a los **hijos** creados por ese shell.

Control de procesos

CTRL + c

finaliza el proceso

CTRL + z

parar un proceso en primer plano el shell toma el control y 'duerme' al proceso que estaba en ejecución

comando &

ejecuta el comando en segundo plano

jobs

muestra información sobre los procesos parados dependientes de este shell

fg

continua un proceso parado, en primer plano. Si existen varios procesos parados, podemos ponerle como argumento el índice que el comando jobs nos mostró.

bg

continua un proceso parado, en segundo plano. Si existen varios procesos parados, podemos ponerle como argumento el índice que el comando jobs nos mostró.

Comandos UNIX

Programación shell

Vamos a realizar pequeños programas

Requisito:

Un editor de texto plano

vi, emacs, joe, nano, pico, mcedit, kwrite, gedit, kate, anjuta, etc.

Mira mamá, soy un script!

Crear un fichero **script.sh** con el siguiente contenido:

```
#!/bin/sh  
echo Mira mamá, soy un script!
```

Intentar ejecutarlo:

```
$ ./script.sh
```

(usamos ./ para indicar que el programa se encuentra en el directorio de trabajo)

Mira mamá, soy un script!

Para que un script se ejecute **debe ser ejecutable** (importante documento)

```
$ chmod +x script.sh
```

Lineas del programa:

Un comentario, desde la aparición de **#** hasta el final de esa linea, donde se indica al shell con la secuencia **#!/bin/sh** qué programa debe usar para ejecutar este fichero.

Un comando que muestra un texto en **stdin**. Es la primera linea "ejecutable", ya que los comentarios son ignorados por el intérprete.

Variables

¿Qué es una **variable**?

Una **variable** es un contenedor.

Consta de:

un identificador que la distingue de otra (su nombre)

un contenido

Por lo general las variables en shell no tienen tipos asociados
(hay distintos tipos de shell)

Variables

Declarar una variable:

```
identificador=contenido
```

Acceder al contenido de una variable:

```
$identificador
```

Ejemplo:

```
$ kleenux=mola ; echo Kleenux $kleenux!
```

Linea de comandos

\$0

contiene el nombre nombre de nuestro script

\$#

el número de parámetros con los que se ha invocado al shell

\$n

los parámetros, con n de 1 a 9 (a **\$#**)

\$\$

el PID de nuestro proceso

\$*

todos los parámetros menos **\$0**

La salida de los programas

Podemos obtener dos cosas distintas de la salida de un programa:

1. su salida en **stdout** y **stderr**
2. el valor **resultado** de su ejecución

\$(comando)

la salida del comando

```
$ salida_ls=$(ls) ; echo $salida_ls
```

\$?

resultado del último programa ejecutado

```
$ ls pirulotropical 2> /dev/null ; echo $?
```

```
$ ls > /dev/null ; echo $?
```

exit ENTERO

termina nuestro programa con el valor de salida ENTERO



Operaciones aritméticas

Si queremos que el shell sustituya una expresión aritmética por su valor:

`$((expresión))`

evalúa la expresión aritmética y reemplaza el bloque por el resultado

sin evaluación:

```
$ echo 1+1
```

con evaluación:

```
$ echo $((1+1))
```

Operadores:

- + suma
- * multiplicación
- resta
- / división entera
- % resto de la división entera
- () agrupar operaciones

Condicionales

Podemos evaluar condiciones gracias a **test**, que es equivalente a []

```
test expresion  
[ expresion ]
```

Este comando evalúa *expresion*, y si evalúa a cierto, devuelve **cero** (true), o en otro caso **uno** (false). Si no hay expresión, test siempre devuelve falso.

Este comportamiento puede ser algo confuso, ya que los valores cierto y falso suelen ser al contrario en otros lenguajes.

Condicionales

Además existen los operadores lógicos:

&& (AND, multiplicación lógica) y

|| (OR, suma lógica),

que se puede aplicar al valor de salida de los programas

Tabla de verdad para **&&** y **||** :

```
$ true && true ; echo $?
```

```
$ true && false ; echo $?
```

```
$ false && true ; echo $?
```

```
$ false && false ; echo $?
```

```
$ true || true ; echo $?
```

```
$ true || false ; echo $?
```

```
$ false || true ; echo $?
```

```
$ false || false ; echo $?
```

if ... then ... [else ...]

Esta es la principal estructura que nos permitirá ejecutar un bloque de código, o (alternativamente) otro, dependiendo de como se evalúe una condición.

```
if CONDICION; then
    bloque de comandos
fi
```

La construcción con **else**:

```
if CONDICION; then
    bloque de comandos b1
else
    bloque de comandos b2
fi
```

if ... then ... [else ...]

Ejemplos:

```
read linea
```

```
# comparamos cadenas de texto, así que usamos comillas
```

```
if [ "$linea" = "secreto" ]; then
```

```
    echo bingo!
```

```
fi
```

```
if ! $(ping -c 1 192.168.0.100 > /dev/null); then
```

```
    echo La máquina 192.168.0.100 no responde
```

```
else
```

```
    echo La máquina 192.168.0.100 está ahí!
```

```
fi
```

Bucles

El shell aporta mecanismos para realizar tareas repetitivas mediante el empleo de estructuras que permiten repetir un bloque de comandos.

Los **bucles**

Existen distintas construcciones de bucles según el shell, vamos a ver dos:

for ... in ...

while ...

for ... in ...

Esta estructura permite **repetir un bloque de comandos** asignando valores de una serie a una variable en cada iteración.

```
for VARIABLE in SERIE; do
    bloque de comandos
done
```

Ejemplos:

```
# equivalente a seq 1 5
for i in 1 2 3 4 5; do
    echo $i
done
```

```
# lo mismo pero con palabras
for palabra in uno dos tres cuatro cinco; do
    echo $palabra
done
```

Rompiendo un bucle: break

En cualquier momento un bucle puede ser interrumpido mediante el uso de **break**, de forma que tras ser ejecutado ese comando el control pasa al siguiente comando después del **fin del bucle**.

Ejemplo:

```
for elemento in *; do
    echo Primer elemento $elemento

    break

    echo Esto nunca se llega a ejecutar
done

echo Seguimos con el programa
```

while ...

Se trata de otra estructura de bucle que permite ejecutar un bloque de comandos **mientras** se evalúe una condición a **cierto**.

```
while CONDICION; do  
    bloque de comandos  
done
```

Ejemplo:

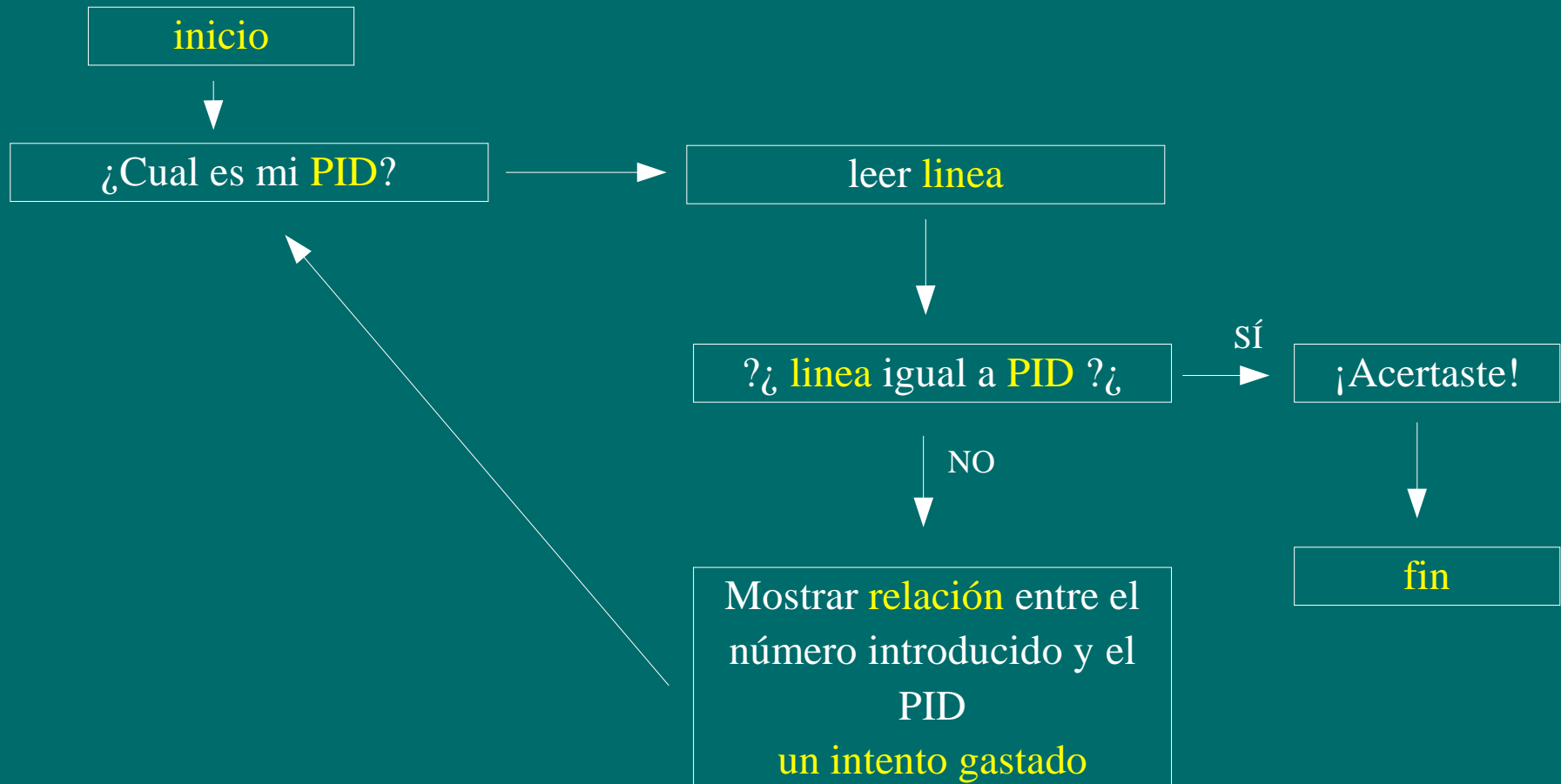
```
# equivalente a seq 1 5  
i=1  
while [ $i -lt 6 ]; do  
    echo $i  
    i=$((i+1))  
done
```

Ejemplo:

```
# lee de stdin hasta que se introduzca 'quit'  
read linea  
  
while [ "$linea" != "quit" ]; do  
    read linea  
done
```

Ejercicio 6: ¿Cual es mi PID?

Estructura del programa:



Gracias por venir

Esperamos que os haya gustado el taller

Más información sobre los próximos talleres en:

<http://www.kleenux.org/>