

Shell scripting

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

15 de marzo de 2021



Universidad  
Rey Juan Carlos



# ¿Qué es la shell?

- Un programa que lee líneas y las ejecuta.
- Las líneas están compuestas de comandos (y más cosas, pipes, redirecciones. . .
- Un comando normalmente tiene argumentos.
- Por ejemplo, el comando `ls -l fichero` tiene dos argumentos.
- Cuando empiezan por - se les suele llamar flags o modificadores.
- Para decirle a un comando que no hay más modificadores, se suele usar `--`.
- Por ejemplo, para borrar un fichero que empieza por -.



## Número mágico

- Tradicionalmente, los primeros bytes de un fichero (normalmente 2 o más), identifican el tipo de fichero (no el nombre).
- Una de las cosas que mira el comando `file`.
- Por ejemplo, JPEG empieza por 0xFF 0xD8.









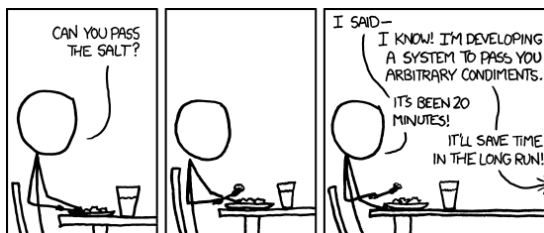


HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?  
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	DAYS	DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	DAYS	DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	DAYS	DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	DAY
	DAY					8 WEEKS	DAYS

(credit <https://xkcd.com>)

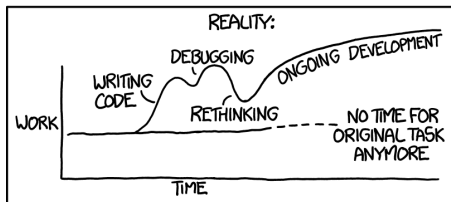
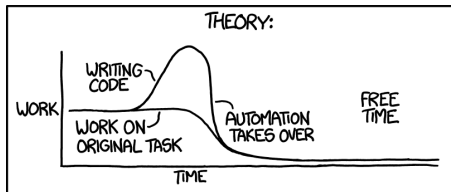
# Automatizar: cuidado



(credit <https://xkcd.com>)

# Automatizar: cuidado

"I SPEND A LOT OF TIME ON THIS TASK.  
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



(credit <https://xkcd.com>)

## Shells

- `sh` es la shell original de Unix, escrita por Ken Thompson. Fue reescrito por Stephen Bourne en 1979 para Unix Version 7: *bourne shell*.
- Los sistemas derivados usan distintas shells: `sh`, `ash`, `bash`, `dash`, `ksh`, `csh`, `tcsh`, `zsh`, `rc`, etc.
- Cada una tiene sus características, pero también tienen mucho en común.
- En sistemas modernos, `/bin/sh` suele ser un enlace simbólico a su shell por omisión para ejecutar scripts. En Ubuntu y Debian es `dash`<sup>1</sup>.
- Política: los scripts que tienen `#!/bin/sh` deben usar únicamente las características POSIX (IEEE Std 1003.1-2017): el subconjunto común que tienen la mayoría de las shells. Así, los scripts pueden ser portables entre distintos sistemas.

<sup>1</sup>No confundir con el shell por omisión para un terminal, que es bash.





# Ejecución de una línea de shell

Pasos:

- 1 Lee la línea, tokeniza
- 2 Hace sustituciones (variables, globbing, etc)
- 3 Abre ficheros de redirecciones
- 4 Ejecuta los comandos















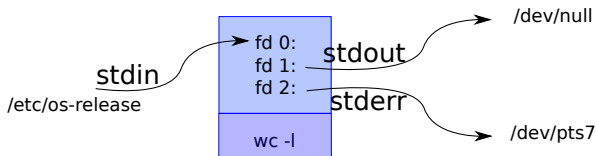
# Redirecciones

- `man 1 bash`, ver la sección *REDIRECTION*

# Redirecciones

- '`<`' y '`>`' abren los ficheros (y crean el de salida si hace falta)
- Y los dejan en la entrada estándar y salida estándar (0 y 1)

```
$ wc -l < /etc/os-release >/dev/null
```



# Redirecciones

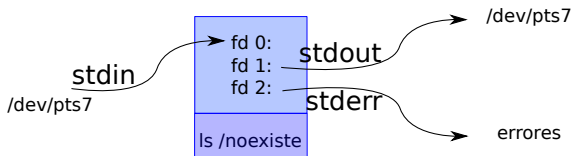
- Para redireccionar la salida de error '2 >' (en general, para entrada o salida con un número delante, se redirecciona ese descriptor de fichero p. ej. '5 <' )

```
$ ls /noexiste 2> errores
```

```
$ cat errores
```

```
ls: cannot access '/noexiste': No such file or directory
```

```
$
```









4

4

4

4







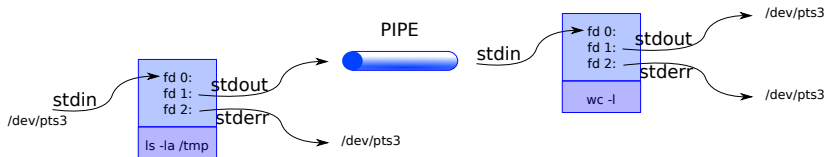
# Pipes

- Mecanismo que conecta dos descriptores de fichero.
- Para conectar la salida (**stdout** o **stderr**) de un proceso con la entrada de otro (**stdin**).
- Línea de comando: pipeline, conjunto de comandos conectados por pipes.
- Concepto de filtro, lee de la entrada, procesa texto, escribe en su salida.

# Pipes

- Lista los ficheros de /tmp, cuenta las líneas de texto de la salida.

```
$ ls -la /tmp/wc -l
40
$
```



## Estado de salida (status)

- Cuando un comando sale, deja el estado de salida (status).
- El estado del último comando que se ejecutó (o pipeline) se puede consultar en \$?
- Es un número, 0 es que ha tenido éxito, un número positivo es un error.
- Se puede (y se debe) hacer salir a un script con el built-in exit que recibe un número como parámetro.



# Ejecución condicional

- `true` y `false`

- Construcciones: `&&` , `||`

```
$ true
```

```
$ echo $?
```

```
0
```

```
$ false
```

```
$ echo $?
```

```
1
```

```
$ true && echo hola
```

```
hola
```

```
$ false && echo hola
```

```
$ true || echo hola
```

```
$ false || echo hola
```

```
hola
```

```
$
```



## Parámetros posicionales

- Se pueden acceder a los parámetros que se han pasado al script con \$1, \$2, \$3 ...
- \$0 expande al nombre con el que se ha invocado el script.
- \$# expande al número de parámetros (sin contar el 0).
- \$\* expande a los parámetros posicionales.
- "\$\*" expande a "\$1 \$2 ..."
- @\$ expande a los parámetros posicionales (igual que \$\* pero separados)
- "\$@" expande a "\$1" "\$2" ...
- shift desplaza los parámetros (p. ej. \$4 pasará a ser \$3). Se actualiza el valor de \$#.
  - Útil para parámetros optativos (pongo lo que sea, o hago shift, el resto igual)



# Ejemplos, parámetros posicionales

```
$ ./param.sh -a fich
$0 es ./param.sh
 $# es 2
$* es -a fich
-a fich
$@ es -a fich
0
```



## Agrupaciones

- Ejecutar en un subshell útil
- Para no cambiar el entorno en el shell actual (`cd`, `export`)

Ejemplo:

```
#sigo en tmp al final:
```

```
$ pwd; (cd /etc; ls apt;); pwd;
```

```
/tmp
```

```
apt.conf.d  sources.list
```

```
preferences.d  sources.list.d  trusted.gpg  trusted.gpg.d
```

/tmp

```
#BLA no existe al final:
```

```
$ echo z$BLA; (export BLA=bla; echo $BLA;); echo z$BLA;
```

Z

bla

Z

\$

# Here documents

- A un programa que lee de su entrada.
- Se le pasa un trozo de texto que se hace pasar por un fichero en su entrada estándar.
- Desde '<<MARCA' hasta una línea que tiene 'MARCA'

```
$ cat <<BLA
soy un
here document
BLA
```

```
soy un
here document
$
```



# Sustitución de comando

- Se sustituye el comando por su salida.
- Se puede escribir de dos formas:

`$(comando)`

`'comando'`

```
$ l=$(wc -l /tmp/a | cut -d' ' -f1)
```

```
$ echo $l
```

```
31
```

```
$
```





## case

Los casos pueden contener patrones de globbing.

```
case palabra in
    patrón1)
        comandos
        ;;
    patrón2 | patrón3)
        comandos
        ;;
    *)    # este es el default
        comandos
        ;;
esac
```

# Bucles

```
while comando
```

```
do
```

```
    comandos
```

```
done
```

```
for variable in palabra1 palabra2 palabraN
```

```
do
```

```
    comandos
```

```
done
```

# Sentencias

- En un script el final de línea tiene significado (acaba el comando).
- Si quiero escribir el if en una línea o de otra forma (por ejemplo, en modo interactivo).
- Puedo poner un punto y coma ';' al final de la sentencia.

Ejemplo:

```
while ls|egrep '\.z$'; do
    comandos
done
```

# Read

- El comando `read` lee una línea de su entrada estándar y la guarda en la variable que se le pasa como argumento.
- Se puede usar para procesar la entrada línea a línea en un bucle.
- Solo debemos hacer eso cuando no tenemos ningún filtro o pipeline que nos sirva para hacer lo que queremos.

# Read

- Por ejemplo, esto itera 2 veces:

```
echo 'a b  
c d' > /tmp/e
```

```
while read line  
do  
    echo $line  
done < /tmp/e
```

- Esto itera 4 veces:

```
for x in $(cat /tmp/e)  
do  
    echo $x  
done
```



- 

uno

dos

tres

```
$ export TFS=-
```

```
$ for i in $(echo uno dos tres) : do echo $i : done
```

uno dos tres

\$

- 

Per esempio:

```
hello () {
```

echo hola \$1

shift

echo adios \$

}

• • •

```
# ejecutamos la función
```

\$ hello uno dos

hola uno

adios dos

\$

+

1

1

# Test

El comando `test` sirve para comprobar condiciones de distinto tipo.

Ficheros:

- -f fichero  
si existe el fichero
- -d dir  
si existe el directorio

## Test

Cadenas:

- `-n String1`  
si la longitud de la string no es cero
- `-z String1`  
si la longitud de la string es cero
- `String1 = String2`  
si son iguales
- `String1 != String2`  
`String1` and `String2` variables no son idénticas
- `String1`  
si la string no es nula

# Test

Enteros:

- Integer1 -eq Integer2  
si los enteros Integer1 e Integer2 son iguales.

### Otros operadores:

- -ne: not equal
- -gt: greater than
- -ge: greater or equal
- -lt: less than
- -le: less or equal

# Test

Test también se puede usar así:

- Esto:

$$[ \quad \$a \text{ -eq} \$b \quad ]$$

- es lo mismo que esto:

```
test $a -eq $b
```







1 2 4

0 2 2

0 0 1

0 1 4

Φ

2-1-4

001

1 0 4

2 2 2

Φ

1-2-4

2 1 4

0 0 1

☐ ☐ ☐  
☐ ☐ ☐

— — —

## Comandos útiles


- 

P. ej:

```
$ diff fich1 fich2
```

```
$ cmp fich1 fich2
```

## Tr

- Traduce caracteres. El primer argumento es el conjunto de caracteres a traducir. El segundo es el conjunto al que se traducen. El enésimo carácter del primer conjunto se traduce por el enésimo carácter del segundo.
- -d   
Borra los caracteres del único conjunto que se le pasa como argumento.
- Se le pueden dar rangos, p. ej.

```
$ cat fichero | tr a-z A-Z
```

## Expresiones regulares (*regex*)



- Es un lenguaje formal para describir/buscar cadenas de caracteres.
- Parecidas a los patrones de la Shell o de globbing, pero más potentes.
- Veremos las que se llaman *extended regular expressions*. Es un estándar de POSIX, `regex(7)`.
- Una string encaja con sí misma, por ejemplo 'a' con 'a'.











# Expresiones regulares (*regex*)

- `exp | exp`  
si encaja con alguna de las regex que están separadas por la barras
- `\`  
carácter de escape: hace que el símbolo pierda su significado especial.

P. ej:

'aass' encaja con la regex `(aass|booo)`

'hola\*' encaja con la regex `a\*`



# Sed

- Stream Editor
- Editor de flujos de texto con comandos.
- Basado en Ed (editor con comandos, tatarabuelo de vi).
- Muchas de las cosas de sed, igual en ed.





## Sed

Ejemplos:

`sed -E '3,6d'` → borra las líneas de la 3 a la 6

`sed -E -n '/BEGIN|begin/,/END|end/p'` → imprime las líneas entre esas regex


`sed -E '3q'` → imprime las 3 primeras líneas.

`sed -E -n '13,$p'` → imprime desde la línea 13 hasta la última.

`sed -E '[Hh]ola/d'` → borra las líneas que contienen 'Hola' u 'hola'.

- `sed -E 's/regex/sustitución/'` → sustituye la primero subcadena que encaja con la exp. por la cadena *sustitución*.
- `sed -E 's/regex/sustitución/g'` → sustituye todas las subcadenas de la línea que encajan con la exp. por la cadena *sustitución*.
- `sed -E 's/(reg)reg(reg).../ \1 sustitución\2/g'`  
→ usa las subcadenas que encajaron con las agrupaciones (los paréntesis en orden de apertura) en la cadena de sustitución. Se llaman referencias hacia atrás o *backreferences*.



`sed -E 's/[0-9]/X/'` -  primer dígito de la línea se sustituye por una X.

`sed -E 's/[0-9]/X/g'` → todos los dígitos de la línea se sustituyen por una X.

sed -E 's/^[A-Za-z]+[ ]+([0-9]+)/NOMBRE:\1 NOTA:\2/g' →  
 añade NOMBRE: y NOTA: delante de los nombres y notas (ojo, no funciona con acentos, guiones, nombres con espacios...).

## hacer mykill.sh

# AWK

- Lenguaje completo de programación de texto.
- Útil, veremos sólo la superficie.
- Al **A**ho, Peter **W**einberger, Brian **K**ernighan.
- Se pueden escribir scripts con AWK como intérprete en el hash bang.

## AWK

- Lee líneas y ejecuta el programa para cada una de ellas.
- No imprime por omisión las líneas que lee.
- Es muy potente, veremos su uso más habitual: imprimir.

## AWK

Imprimir:

- `print`  
Sentencia que imprime los operandos. Si se separan con comas, inserta un espacio. Al final imprime un salto de línea.
- `printf()`  
Función que imprime, ofrece control sobre el formato de forma similar a la función de libc para C:

```
$ ls -l | awk '{ printf("Size:%08d KBytes\n", $5) }'
```

Variables:

- \$0  
La línea que está procesando.
- \$1, \$2 ...  
El primer, segundo... campo de la línea.
- NR  
Número del registro (línea) que se está procesando.
- Ejemplo  
para imprimir la tercera y segunda columna de un csv:  

```
$ cat a.txt | awk -F, '{printf("%d%d", $3, $2)}'
```



patrón { programa }

Actuando sólo en unas líneas, que se ajustan a un patrón, que puede ser:

- Expresión regular

Se procesan las líneas que encajen con la regex.

```
$ ls -l | awk '/[Dd]esktop/{ print $1 }'
```

```
$ ls -l | awk '$1 ~ /[Dd]esktop/ { print $1 }'
```

# AWK

- Expresión de relación

Se comparan valores y se evalúa la expresión.

```
$ ls -l | awk ' NR >= 5 && NR <= 10 { print $1 }'
```



# AWK

Inicialización y finalización:

```
BEGIN{
...
}
```

```
patrón{
  ...
}
```

$$\begin{array}{l} \text{END}\{ \\ \quad \cdot \cdot \cdot \\ \} \end{array}$$

# AWK

- next: pasar a la siguiente regla

# AWK

Arrays asociativos:

- Son cómodos.
- Por ejemplo, para imprimir cuantos procesos tiene ejecutando cada usuario en el sistema:

```
$ ps aux | awk '{dups[$1]++} END{for (user in dups) {print user,dups[user]}}'
```

# Recorrer un árbol

- Para recorrerse un árbol de ficheros
  - `du -a .`
  - `find .`



## Join

- **join** quita las que no están en alguno de los dos (inner join)
- Tienen que estar ordenadas, usar sort antes
- Igual que sort puede usar diferentes campos

# Xargs

- Sirve para usar como argumentos en la ejecución de otro comando lo que viene por la entrada estándar.

```
$ echo a b c | xargs ls -l
```

```
-rw-rw-r-- 1 esoriano esoriano 2 mar  1 16:03 a
-rw-rw-r-- 1 esoriano esoriano 2 mar  1 16:03 b
-rw-rw-r-- 1 esoriano esoriano 2 mar  1 16:03 c
```

