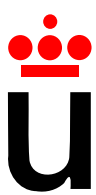
Shell scripting

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

4 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 `-`.

Dos formas de ejecutar la shell

- Modo interactivo, como hasta ahora, escribo comandos, los ejecuta.
- Modo batch, en un script, es un intérprete más.
- En Unix hay un mecanismo especial para esto.
- Primero hay que entender el número mágico en Unix.

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`.

Hash bang: #!

- En particular los ejecutables, si empiezan por los dos caracteres ASCII `#!` (`0x23 0x21`) son interpretados.
- Eso significa que hay un comando externo que interpreta el fichero.
- Se ejecuta el comando con el fichero como último parámetro.
- Esto lo usa, Python, los scripts de shell...

```
 $ cat hh
#!/bin/echo --
esto da igual
 $ chmod +x hh
 $ ./hh
-- ./hh
```

Scripts de shell

- Son programas interpretados mediante el mecanismo *hash bang* que usan la shell como intérprete.
- Para depurar, se puede poner `-x` como parámetro para que la shell escriba el comando antes de ejecutarlo.

```
#!/bin/sh -x

echo hola mundo
pwd
exit 0
```

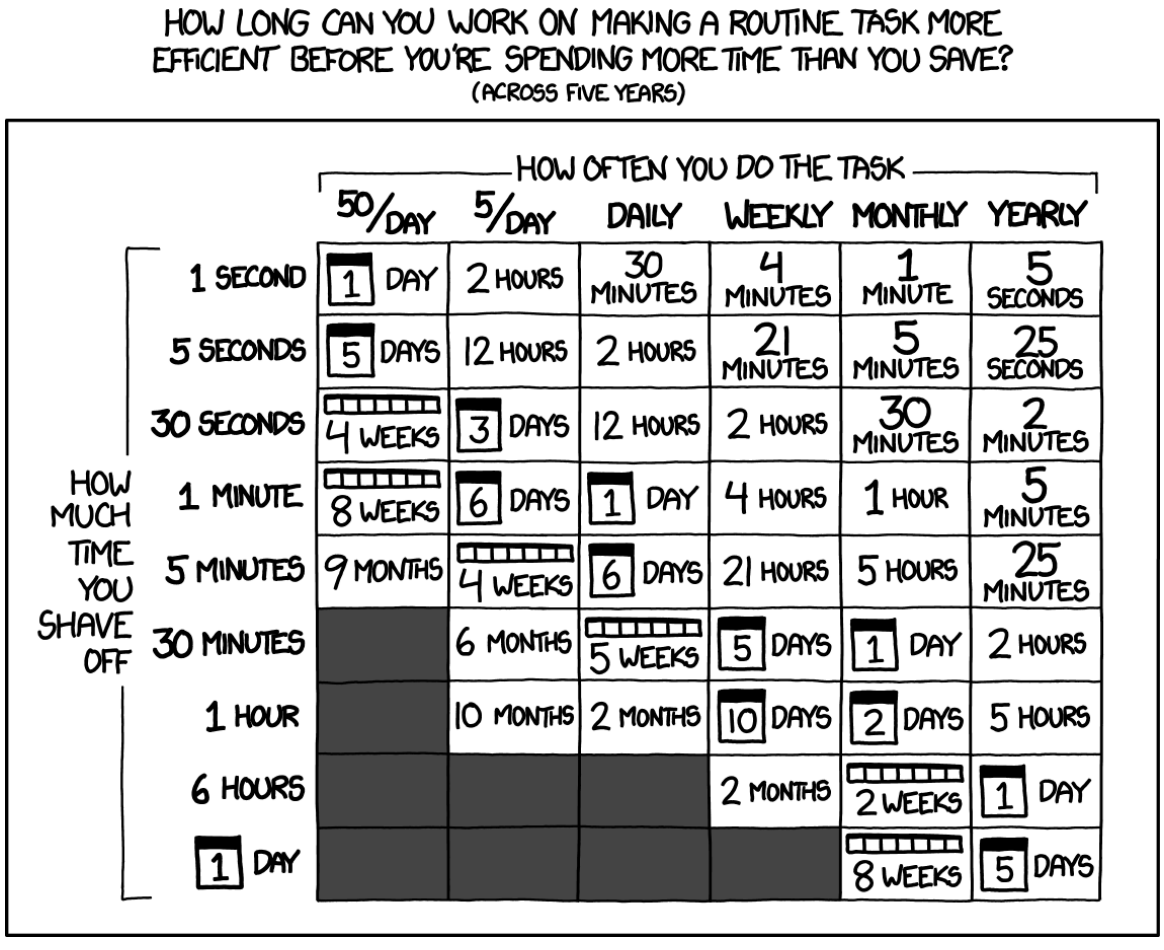
¿Cuándo hago un script de Shell?

- Pasos para realizar una tarea:
  1. Mirar si hay alguna herramienta que haga lo que queremos → buscar en el manual.
  2. Si no encontramos, intentar combinar distintas herramientas → **Linea de comandos**. La primera aproximación debe ser pipelines de filtros, etc.
  3. Si es complicado, intentar combinar todavía más distintas herramientas → **programar un script de Shell**. La primera aproximación debe ser pipelines de filtros, etc. Esto además permite reutilizar el trabajo.
     IDEA: combinar herramientas que hacen bien una única tarea para llevar a cabo tareas más complejas.
  4. Si no podemos, hacer una herramienta → programada en C, Python, Java, Ada, Go, ...

¿Qué tipo de cosas ?

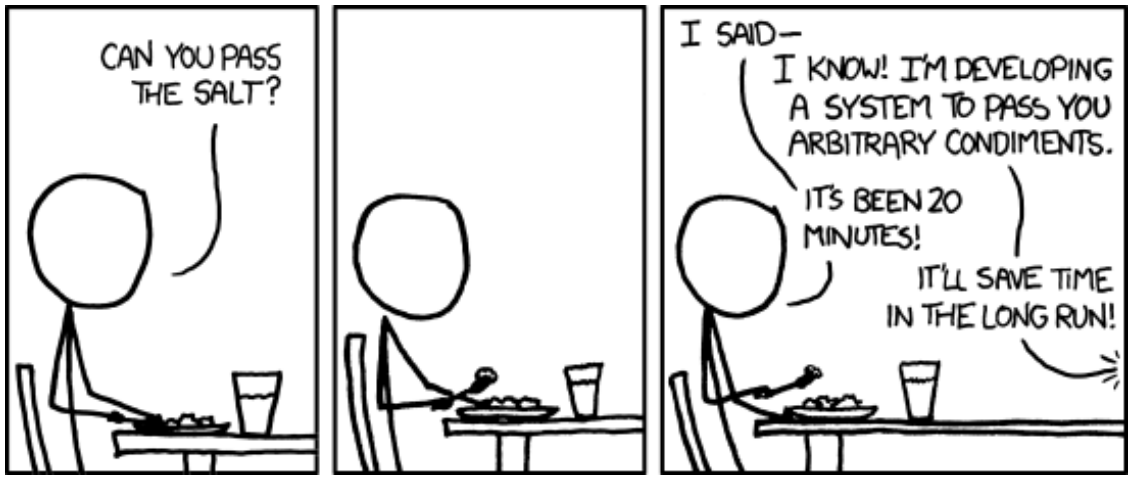
- La shell es especialmente buena
  - Para tareas que hago una vez
  - Para automatizar tareas (con un IDE, Makefile)
  - Para procesar texto

Automatizar



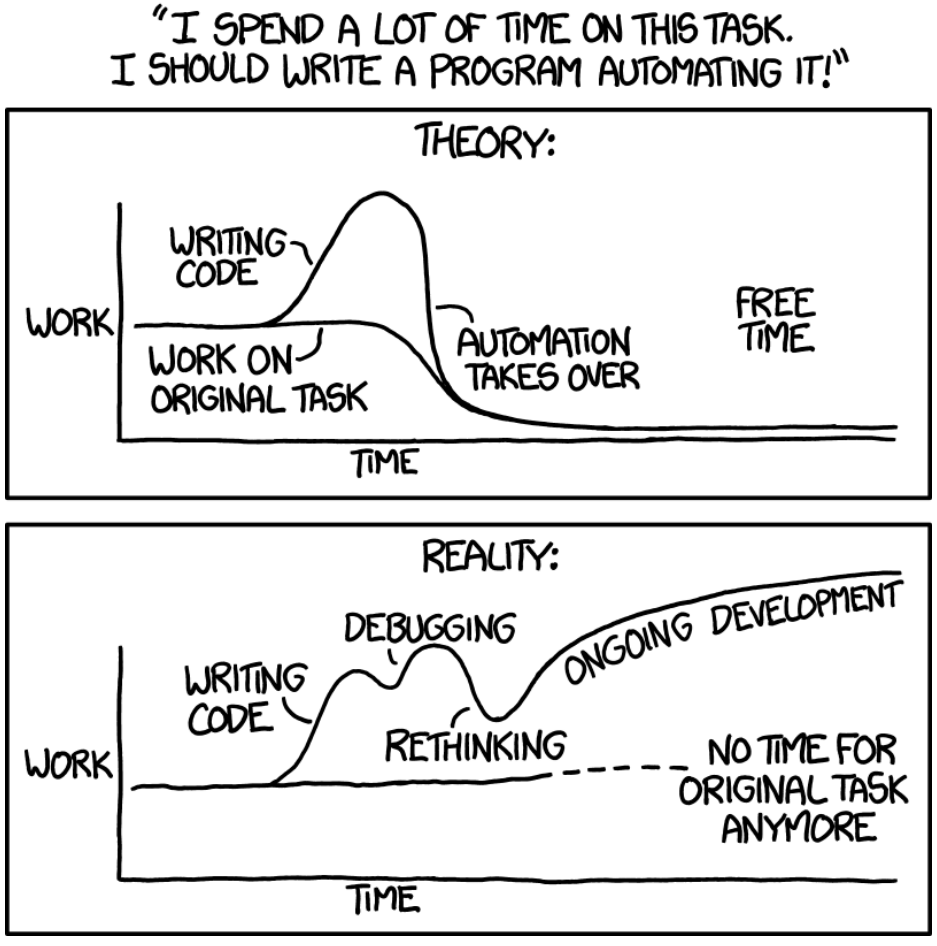
(credit `https://xkcd.com`)

Automatizar: cuidado



(credit `https://xkcd.com`)

Automatizar: cuidado



(credit `https://xkcd.com`)

Shells

- `sh` es la shell original de Unix, escrita por Ken Thompson. Fue rescrito 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`[1].
- 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.

[1] No confundir con el shell por omisión para un terminal, que es bash.

Un script:

- Tiene que tener permisos de ejecución.
- Hay comandos que se implementan dentro del shell (no se ejecuta un fichero externo al shell, es una parte del propio shell). Se llaman *built-in*.
- El comando *built-in* `exit` sale del script con el status indicado en su argumento.
- Si un script no sale con `exit`, deja el status que tiene `$?`.

```
#!/bin/sh

# este es un hola mundo en sh
# esto es un comentario

echo hello world
exit 0
```

Un script:

- Una ventaja de la shell, es que puedo probar de forma interactiva
- No escribo el script directamente, voy probando los comandos
- O ni siquiera escribo el script (escribo los comandos directamente)

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

Variables

- Se definen con el built-in infijo $=$.
- Definirlas las cambia en la shell, pero no las exporta a los hijos.
- Es importante recordar que la sustitución la hace la shell.

```
 $ x=bla
 $ echo $x
bla
 $ touch c d e
 $ ls
c  d  e
 $ cmd=ls
 $ $cmd
c  d  e
 $
```

Variables

- Forman parte del entorno.
- El entorno lo hereda el proceso del padre.
- El comando built-in `export` sirve para meter a una variable en el entorno y que se herede.

```
 $ hola=bla
 $ echo $hola
bla
 $ bash -c 'echo $hola'

 $ export hola
 $ bash -c 'echo $hola'
bla
 $
```

Entrada estándar, salida estándar

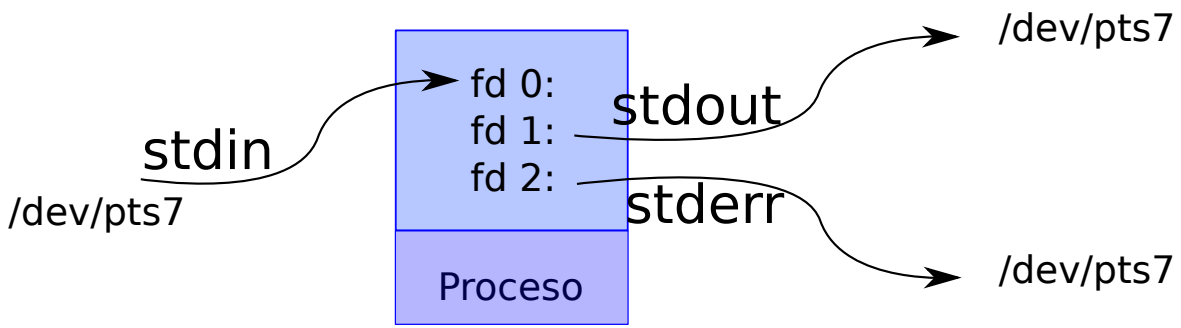
- En Unix, los ficheros abiertos de un proceso están numerados (File Descriptor, FD o *descriptor de fichero*), cada vez que abro uno para usarlo: un número.
- Hay tres que vienen abiertos por defecto entrada y salida estándar y de error, 0 (**stdin**), 1 (**stdout**), 2 (**stderr**).
- Por defecto es la consola la que viene abierta en esos descriptores (cuando ejecutamos un shell).

Entrada estándar, salida estándar

- Stdin es de donde lee datos por defecto el programa (entrada estándar).
- Stdout es dónde escribe datos (salida estándar).
- Stderr es dónde escribe mensajes de error (salida de error).
- Este sistema permite mandar la salida de un comando a un fichero o leer de él o conectar varios comandos.
- Stderr permite escribir en la salida mensajes de error sin interferir con los datos.

Entrada estándar, salida estándar

- Por defecto, viene abierta la consola.



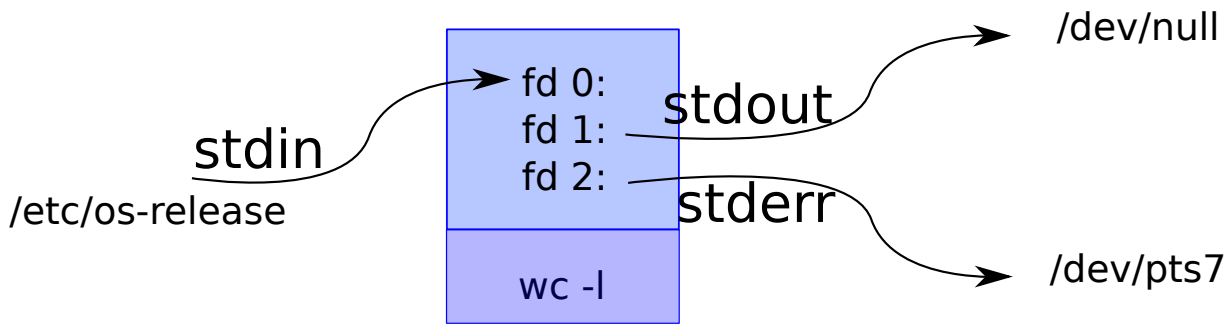
Redirecciones

- `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)

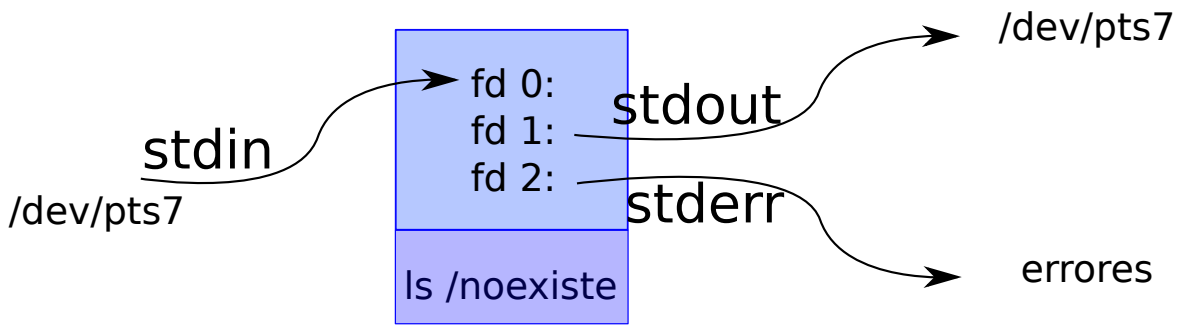
```
$ wc -l < /etc/os-release >/dev/null
```



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 <')

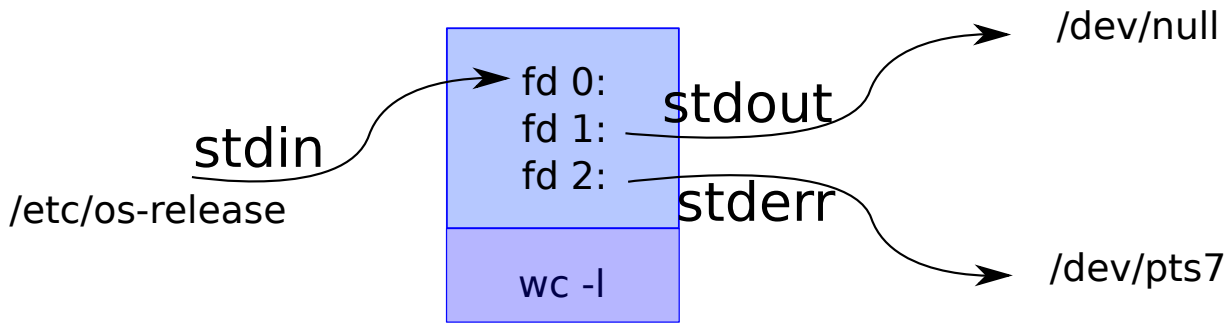
```
$ ls /noexiste 2> errores
$ cat errores
ls: cannot access '/noexiste': No such file or directory
$
```



Redirecciones

- '>>' hace lo mismo que '>'
- Pero *append* (añade al final), mientras que '>' trunca el fichero.

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



Redirecciones

- Para mandar la salida de error al mismo sitio que la estándar o viceversa (ojo, el orden importa).
- es $n > \&m$

Salida de error al mismo fichero:

```
$ ls /noexiste > fich 2>&1
$ cat fich
ls: cannot access '/noexiste': No such file or directory
$
```

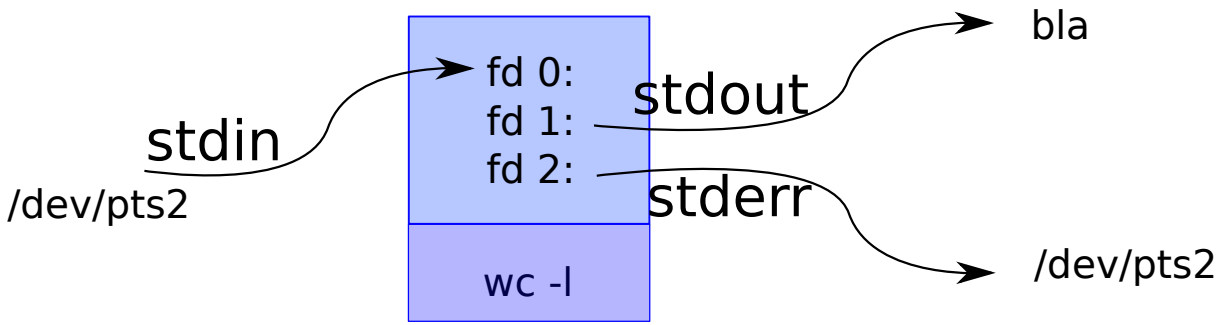
Escribir algo por la salida de error de un script:

```
echo aaa 1>&2
```

OJO

- '>' trunca el fichero antes de ejecutar el comando.
- Recordar el orden en el que hace las cosas las shell.
- Puedo acabar con el fichero vacío.

```
$ clear
$ echo aaaaaaa > bla
$ cat bla
aaaaaaa
$ cat bla > bla
$ cat bla
$
```



Inspección

- Puedo usar el sistema de ficheros especial sintético (no son ficheros de verdad) `/proc`
- `ls -l /proc/PID/fd`

```
 $ echo $$
14927
 $ ls -la /proc/$$/fd
total 0
dr-x------ 2 paurea paurea  0 feb 11 15:45 .
dr-xr-xr-x 9 paurea paurea  0 feb 11 15:45 ..
lrwx------ 1 paurea paurea 64 feb 11 15:45 0 -> /dev/pts/9
lrwx------ 1 paurea paurea 64 feb 11 15:45 1 -> /dev/pts/9
lrwx------ 1 paurea paurea 64 feb 11 15:45 2 -> /dev/pts/9
lrwx------ 1 paurea paurea 64 feb 11 15:49 255 -> /dev/pts/9
 $
```

Ficheros especiales

- Ficheros sintéticos servidos por el kernel, sin respaldo en almacenamiento (como los de `/proc`)
- Fichero "agujero negro" todo lo que mande se pierde: `/dev/null`
- Fichero "fuente" da bytes a cero: `/dev/zero`

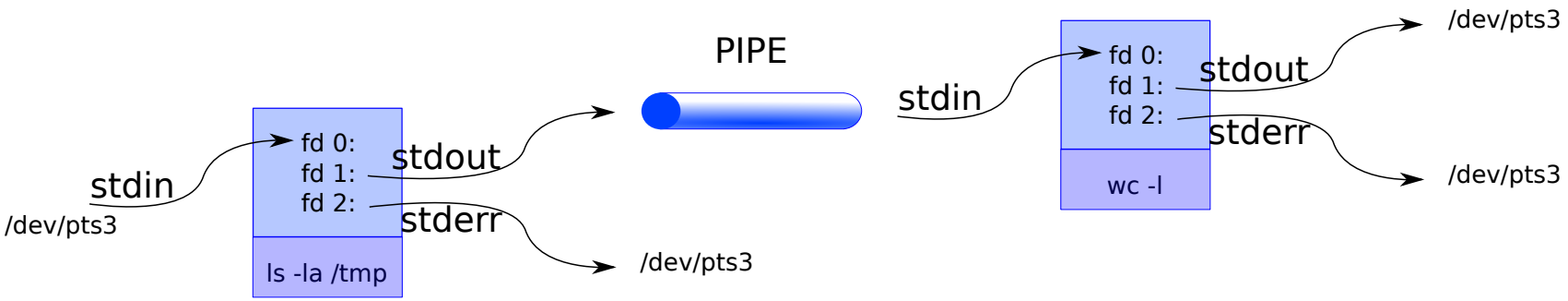
Pipes

- 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
 $
```

Hemos visto:

- | es un pipe
- & ejecuta un comando en background
- $ se usa para las variables. $var es lo mismo que ${var}
- " y ' se usan para escapar cadenas (las dobles expanden algunas cosas)
- < , < y >> son redirecciones
  - 5> para redirigir el fd 5
  - 5>&3 para redirigir duplicando el fd 3 en el 5.
  - Cuando hay varios el orden importa ej: `2>&1 >/dev/null`
- \ se usa para escapar caracteres
- && , || para ejecución condicional
- Globbing (wildcards): ? * $[a-z]$ etc.

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

```
 $ cat param.sh
#!/bin/sh

echo \$0 es $0
echo \$\# es $#
echo \$\* es $*

echo $1 $2 $3 $4
echo \$\@ es $@
shift
shift
echo $1 $2 $3 $4 $#
 $ ./param.sh -a -b -c fich
$0 es ./param.sh
$# es 4
$* es -a -b -c fich
-a -b -c fich
$@ es -a -b -c fich
-c fich 2
```

Ejemplos, parámetros posicionales

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

Agrupaciones

- Si queremos ejecutar comandos en un subshell:

(comando; comando; ...)

- Si queremos ejecutar una agrupación de comandos en el shell actual:

{ comando; comando; ... }

Ejemplo:

```
$ { echo uno; echo dos; } | tr o 0
un0
d0s
$ { echo los ficheros de /tmp son; ls /tmp; } > ficheros
```

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
$
```

if

Las condiciones depende del status de salida del comando: éxito es verdadero, fallo es falso.

```
if comando
then
        comandos
elif comando
then
        comandos
else
        comandos
fi
```

if

Se puede negar la condición del resultado de un comando con la admiración.

```
if ! comando
then
        comandos
fi
```

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
```

Variable IFS

- Esta variable contiene los caracteres que se usan como separadores entre campos.
- Por omisión contiene el tabulador, espacio y el salto de línea.
- Hay que tener cuidado: cambiar el valor de esta variable rompe las cosas.
- Mejor en subshell.

```
 $ for i in $(echo uno dos tres) ; do echo $i ; done
uno
dos
tres
 $ export IFS=-
 $ for i in $(echo uno dos tres) ; do echo $i ; done
uno dos tres
 $
```

Funciones

- Se pueden definir funciones. Se accede a sus parámetros como a los parámetros posicionales.
- Se llama como un comando

Por ejemplo:

```
hello () {
        echo hola $1
        shift
        echo adios $1
}

...
# ejecutamos la función
 $ hello uno dos
hola uno
adios dos
 $
```

Alias

- Similar a funciones.
- Sin argumentos, `alias` muestra los que hay definidos.
- Para la línea de comandos

```
 $ alias hmundo='echo hola mundo'
 $ hmundo
hola mundo
 $ unalias hmundo
 $ hmundo
hmundo: command not found
 $  alias
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
 $
```

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:

```
[  $a -eq $b  ]
```

- es lo mismo que esto:

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

Operaciones aritméticas

Para operaciones básicas con enteros podemos usar el propio shell. También podemos usar el comando `bc`.

- Esto:

```
$((5 + 7))
```

- se reemplaza por

```
12
```

Filtros útiles

- `sort`
  ordena las líneas de varias formas.
- `uniq`
  elimina líneas contiguas repetidas.
- `tail`
  muestra las últimas líneas.

P. ej:

```
$ ps | tail +3 # a partir de la 3ª
$ ps | tail -3 # las 3 ultimas
$ seq 1 1000 | sort
$ seq 1 1000 | sort -n
```

Sort

- Puede recibir una lista de columnas (empezando por la 1)
- Y un separador
- Y ordena por esos campos como clave (es un intervalo de campos)
- Ojo con estabilidad (`-s`)

```
 $ cat x.txt
1-2-4
2-3-3
2-2-1
2-1-4
 $ sort -k2,2 -t\- x.txt
2-1-4
2-2-1
1-2-4
2-3-3
 $  sort -k1,2 -t\- x.txt
1-2-4
2-1-4
2-2-1
2-3-3
```

Comandos útiles

- `diff`
  compara ficheros de texto línea a línea
- `cmp`
  compara ficheros binarios byte a byte

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 (*regexp*)

- 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 estádar de POSIX.
- Una string encaja con sí misma, por ejemplo `'a'` con `'a'`.

Expresiones regulares (*regexp*)

- `.`
  encaja con cualquier carácter, por ejemplo 'a'.
- `[conjunto]`
  encaja con cualquier carácter en el conjunto, por ejemplo `[abc]` encaja con `'a'`. Se pueden especificar rangos, p. ej. `[a-zA-Z]`.
- `[^conjunto]`
  encaja con cualquier carácter que **no esté** en el conjunto, por ejemplo `[^abc]` NO encaja con `'a'`, sin embargo sí encaja con `'z'`.

Expresiones regulares (*regexp*)

- ^
  encaja con *principio de línea*.
- $
  encaja con *final de línea*.
- Una regexp e_1 concatenada a otra regexp e_2, $e_1 e_2$, encaja con una string si una parte p_1 de la string encaja con e_1 y otra parte contigua, p_2, encaja con e_2.

P. ej:
'az' encaja con la regexp `[a-d]z`

Expresiones regulares (*regexp*)

- `exp*`
  encaja si aparece **cero o más veces** la regexp que lo precede.
- `exp+`
  encaja si aparece **una o más veces** la regexp que lo precede.

P. ej:

`'aaa'` encaja con la regexp `a*`

`'baaa'` encaja con la regexp `ba+`

`'bb'` encaja con la regexp `ba*`

`'bb'` no encaja con la regexp `ba+`

Expresiones regulares (*regexp*)

- `exp?`
  encaja si aparece **cero o una vez** la regexp que lo precede. Se utiliza para partes opcionales.
- `(exp)`
  agrupa expresiones regulares.

P. ej:

`'az', 'av', 'a'` encajan con la regexp az?

`'abab'` encaja con la regexp `(ab)+`

`'abab', 'ababab', 'abababababa'` encajan con la regexp `(ab)+`

Expresiones regulares (*regexp*)

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

P. ej:

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

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

Egrep

- Filtra líneas usando expresiones regulares.
- `-v`
  Realiza lo inverso: imprime las líneas que no encajan.
- `-n`
  Indica el número de línea.
- `-e`
  indica que el siguiente argumento es una expresión.
- `-q`
  silencioso, no saca nada por la salida (cuando solo nos interesa el status de salida).

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

- Es un editor: aplica el comando de sed a cada línea que lee y escribe el resultado por su salida. Sin el modificador `-n`, escribe todas las líneas después de procesarlas.
- Si queremos usar expresiones regulares extendidas, hay que usar la opción -E.
- Comandos:
  `q` → Sale del programa.
  `d` → Borra la línea.
  `p` → Imprime la línea. (correr con -n)
  `r` → Lee e inserta un fichero.
  `s` → Sustituye. ← la que más se usa!!!

Sed

- Direcciones:
  `número` → actúa sobre esa línea.
  `/regexp/` → líneas que encajan con la regexp.
  `$` → la última línea.
- Se pueden usar intervalos:
  `número,número` → actúa en ese intervalo.
  `número,$` → desde la línea *número* hasta la última.
  `número,/regexp/` → desde la línea *número* hasta la primera que encaje con regexp.

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 regexp

`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

Sustitución

- `sed -E 's/regexp/sustitución/'` → sustituye la primero subcadena que encaja con la exp. por la cadena *sustitución*.
- `sed -E 's/regexp/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

Ejemplos

`sed -E 's/[0-9]/X/'` → el 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) }'
```

AWK

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)}'
```

AWK

Variables:

- NF
  Número del campos del registro que se está procesando.
- var=contenido
  Se pueden declarar variables dentro del programa. Con el modificador -v se pueden pasar variables al programa.

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

AWK

`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 regexp.

```
$ 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{
...
}

END{
...
}
```

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 recorrese un árbol de ficheros
  - `du -a .`
  - `find .`

Join

- `join`
- Extremadamente útil
- Hace un *join* relacional de dos columnas (tienen que estar ordenadas)

```
$  echo '
a bla
b ble
c blo' > a.txt
$  echo '
a ta
b te
c to' > b.txt
$  join a.txt b.txt
a bla ta
b ble te
c blo to
```

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
```

Comandos texto

- `cut` y `paste`
- Mejor dominar `awk` (se puede hacer todo)
- Ejemplos:

```
 $ echo uno dos tres  | cut -d' ' -f 1,3
uno tres
 $ ps  | sed 3q > a
 $ seq 1 3 > b
 $ paste a b
PID TTY          TIME CMD        1
8462 pts/4    00:00:00 bash       2
11357 pts/4    00:00:00 ps        3
 $  paste b a
1          PID TTY          TIME CMD
2         8462 pts/4    00:00:00 bash
3        11357 pts/4    00:00:00 ps
```