La Shell I

Escuela Técnica Superior de Ingeniería de Telecomunicación

gsyc-profes (arroba) gsyc.es

Diciembre de 2020

Contenidos

- La shell más habitual es *bash*, pero hay muchas otras *sh*, *csh*, *dash*
- Las **órdenes** generalmente son solo pequeños programas ejecutables
- El nombre original es *shell command*. En español puede decirse *comando*, *orden* o *mandato*.

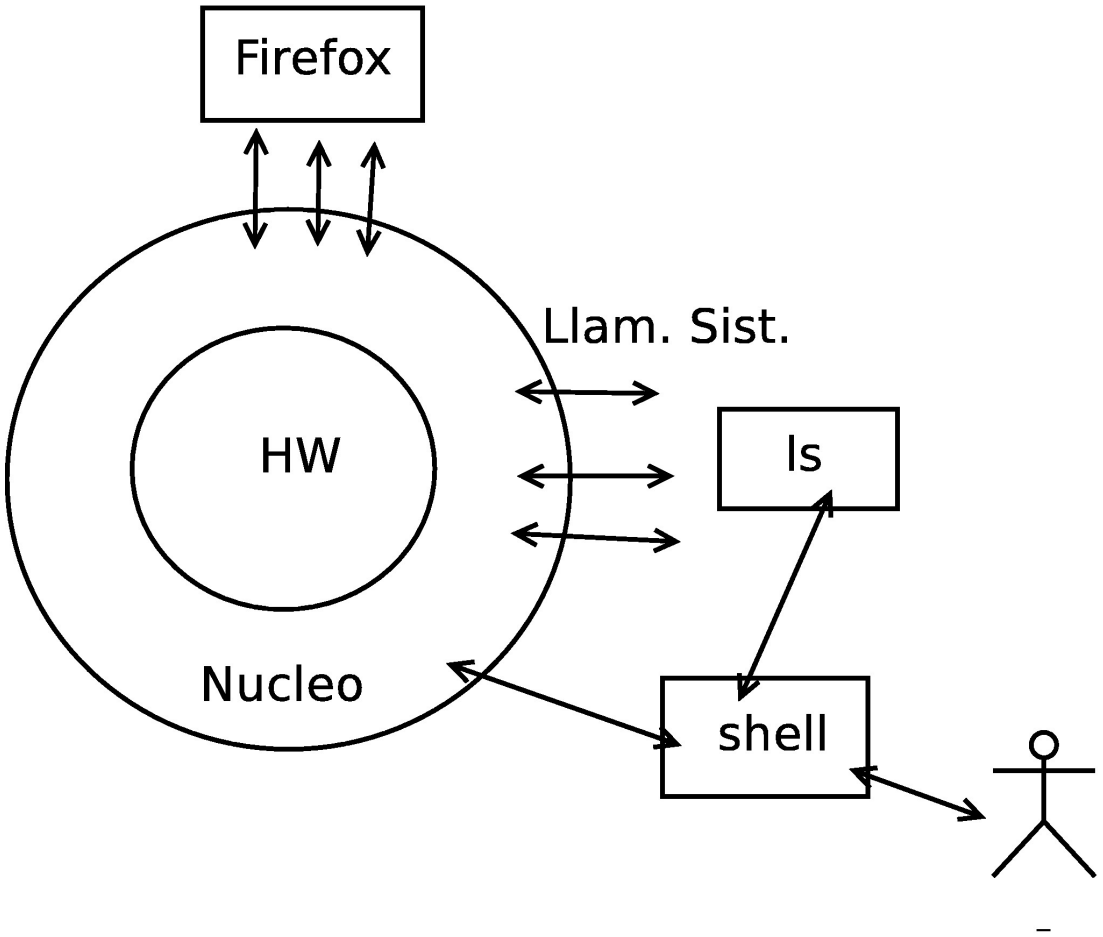


Figura: El Sistema Operativo

¿Quién soy? ¿Dónde estoy? ¿Qué tengo?

- `whoami`
  Muestra el usuario
- `id`
  Muestra usuario y grupos
- `uname`
  `uname -a`
  Versión de Linux
- `hostname`
  Nombre de máquina
- `pwd`
  Directorio de trabajo actual
- `w`
  Usuarios conectados a la máquina

- `du` Espacio de disco ocupado por los ficheros de un directorio
  `du -s` Espacio de disco ocupado por un directorio
  `du -h` Unidades legibles para un humano
- `df`
  Espacio de disco libre
- `lsblk -f`
  Listado de todos los discos (dispositivos de bloques)

- `ls -l` Formato largo
  `ls -a` Muestra ficheros ocultos (empiezan por punto)
  `ls -lh` Formato largo, unidades legibles por humano
  `ls -R` Recursivo
  `ls -ld` Lista el directorio, no su contenido

Unix es *case sensitive*

Metacaracteres de la Shell

- `$` Variable
- `*` 0 o más caracteres cualquiera
- `?` exactamente 1 carácter cualquiera
- `[]` 1 carácter de la clase

ejemplo:

```
ls *.txt
```

el shell lo expande a

```
ls  texto1.txt texto2.txt texto3.txt
```

La orden recibe 3 argumentos, no sabe nada de metacaracteres

Funcionamiento (simplificado) de la shell

La shell:

1. Lee texto de fichero stdin (por ejemplo, el teclado). Aporta algunas facilidades al usuario (borrar, autocompletar)
2. Analiza el texto (expande metacaracteres y variables)
3. Toma la primera palabra y busca una orden con ese nombre en los directorios indicados por PATH
4. Si puede, ejecuta la orden y se queda dormida esperando a que acabe
   Por ejemplo
   ```
   koji@mazinger:~$ xcalc
   ```
   (Mientras usamos la calculadora, la shell permanece inactiva)
   - Si queremos que la shell siga activa, lanzamos el proceso en segundo plano (*background*)
     ```
     koji@mazinger:~$ xcalc&
     ```

- Una aplicación lanzada sin `&`, se dice que está lanzada en primer plano (*foreground*).
- La shell se cierra con la orden `exit`. (O con `ctrl d`, que representa el fin de fichero)

Autocompletado

Con frecuencia pasaremos a los mandatos nombres de fichero (como argumento). La función de autocompletar evita teclear nombres completos
Supongamos que tenemos dos ficheros en el directorio actual

```
.
|-- mi_fichero_del_martes
`-- un_fichero_ejemplo
```

No es necesario teclear

```
koji@mazinger:~$ ls -l mi_fichero_del_martes
```

Como solo hay un fichero que empiece por *mi*, basta escribir

```
koji@mazinger:~$ ls -l mi
```

y luego pulsar tab

Si hay más de un fichero que empiece por *mi*

```
.
|-- mi_fichero_del_martes
|-- mi_fichero_del_miercoles
`-- un_fichero_ejemplo
```

```
koji@mazinger:~$ ls -l mi_fichero_del_m
mi_fichero_del_martes     mi_fichero_del_miercoles
```

Autocompletar rellena hasta donde puede, nos ofrece los ficheros que encajan en lo que hemos escrito, y espera a que introduzcamos una letra más para deshacer la ambigüedad (en este ejemplo, 'a' o 'i')

La shell también autocompleta nombres de ejecutables (si tienen permiso de ejecución y están en el path)

```
koji@mazinger:~$ pass<TAB>
```

Se autocompleta a

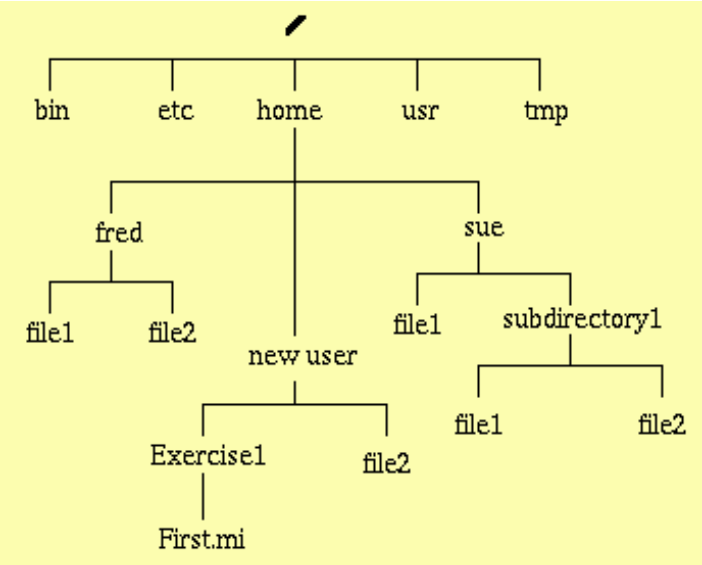
```
koji@mazinger:~$ passwd
```

De esta manera no hace falta teclear todas las letras. Ni recordar el nombre exacto de órdenes largas, basta saber cómo empiezan

history

La shell recuerda las últimas órdenes ejecutadas. Podemos desplazarnos sobre ellas con los cursores arriba/abajo

Árbol de directorios



- Árbol, todo cuelga de un único directorio raiz
- Dentro de cada directorio, habrá ficheros o subdirectorios
- jerarquía clásica unix:
  - /home
  - /bin
  - /usr
  - (...)

Nombres de fichero

- Hasta 256 caracteres
- Mayúsculas y minúsculas son distintas
  - Se puede tener en un mismo directorio los ficheros `ejemplo`, `EJEMPLO` y `EjemPlO`
  - Pero si llevamos estos ficheros a una unidad externa (pendrive, disco) que mantenga su formato por omisión (FAT32), deja de ser legal
- Los que empiezan por punto (`.`)se consideran ocultos (por defecto no se muestran), suelen usarse para ficheros o directorios de configuración
- Casi cualquier carácter es legal, pero es preferible usar solo números, letras, guión y barra baja.
  - Es preferible evitar los espacios
  - También es buena idea evitar eñes y tildes (Naturalmente, hablamos del nombre del fichero, no de su contenido)

Permisos

ls -l: Muestra los contenidos de los directorios en **formato largo**:

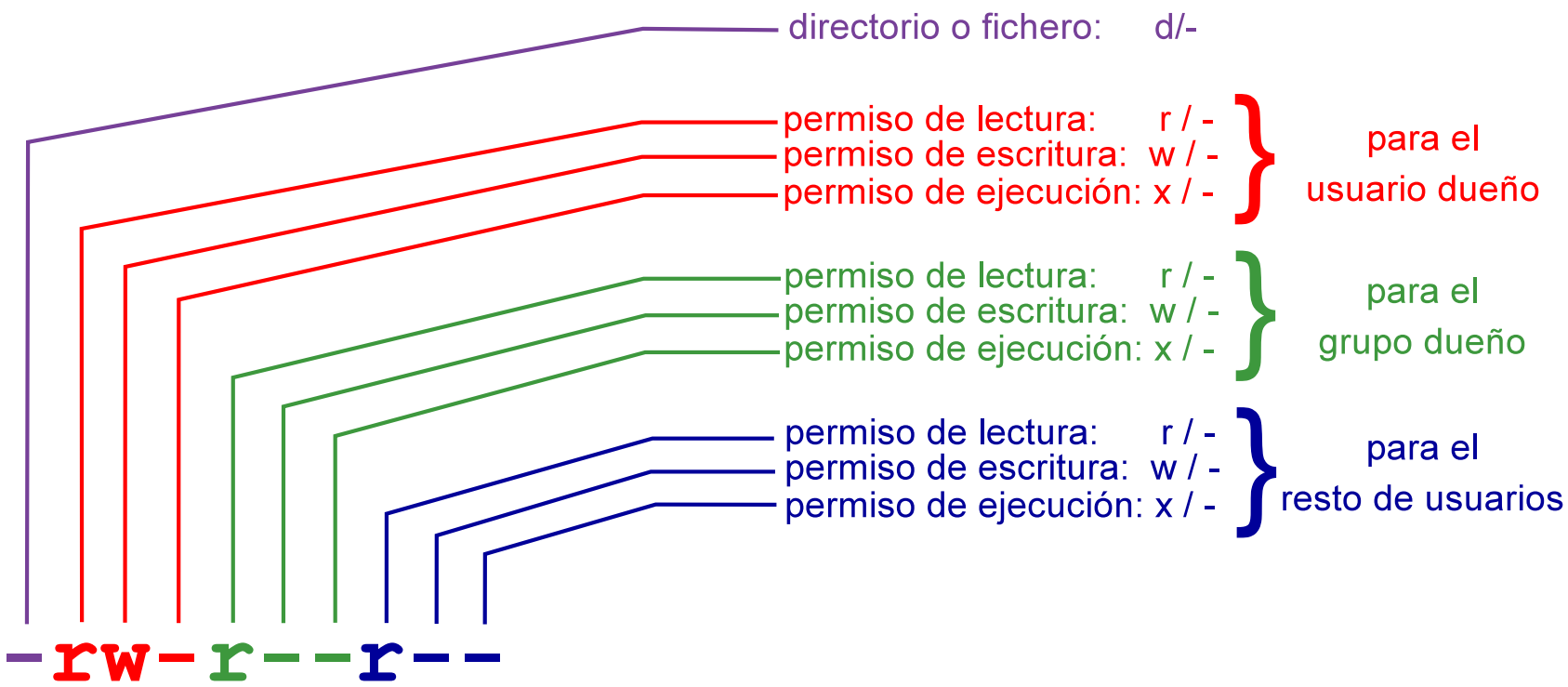
```
drwxr-xr-x 2 jperez al-07-08 4096 2007-10-09 22:51 d1
-rw-r--r-- 1 jperez al-07-08 8152 2007-10-16 09:42 f1
-rw-r--r-- 1 jperez al-07-08   24 2007-10-16 09:42 f3
```

El primer carácter indica:

```
-       Regular file - Fichero ordinario
d       Directory - Directorio
l       (Symbolic) Link - Enlace simbólico
p       Named pipe - Pipe con nombre
s       Socket - Socket
c       Character device - Dispositivo orientado a carácter
b       Block device - Dispositivo orientado a bloque
```

Para cada entrada, aparece, además:

- permisos: Los 10 primeros caracteres
- número de nombres del fichero (enlaces duros)
- usuario del dueño
- grupo del dueño
- tamaño en bytes
- fecha y hora de la última modificación
- nombre



NOTA: En inglés, al conjunto de permisos de un fichero se le conoce como el "modo de acceso" (*access mode*).

- Permisos de un fichero:
  - El de **lectura**: permite ver su contenido
  - El de **escritura**: permite modificar su contenido
  - El de **ejecución**: permite ejecutarlo
- Permisos de un directorio:
  - El de **lectura**: permite hacer `ls` del contenido
  - El de **escritura**: permite crear y borrar ficheros y subdirectorios dentro de él
  - El de **ejecución**: permite hacer `cd` a él

Permisos *normales* de un fichero: `-rw-r--r--`
Permisos *normales* de un directorio: `drwxr-xr-x`

Cambio de permisos

- Los permisos se representan con una secuencia de caracteres: r,w,x (lectura, escritura y ejecución)
- Un guión indica la ausencia del permiso correspondiente a esa posición

Para cambiar permisos se usa `chmod`, que tiene dos sintaxis equivalentes, se puede usar la que resulte más cómoda

1. `chmod 754 mi_fichero`
   No importan los permisos que tuviera previamente el fichero, pasa a tener:

```
  7   5   4         (octal)
111 101 100         (binario)
rwx r-x r--
```

2. `chmod [ugo] [+-] [rwx] mi_fichero`
   `chmod o+x mi_fichero`
   A partir de los permisos que tuviera el fichero, se suman o se restan los permisos indicados a u,g,o (user, group, other)

- Para la primera forma, es necesario saber contar en binario hasta 7

Octal	Binario
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Permisos de los directorios

`chmod -R` Cambia permisos recursivamente

- `r` y `x` normalmente van juntos. (Ambos o ninguno). Permiten entrar en el directorio y listar
- `w` permite añadir añadir ficheros o borrarlos

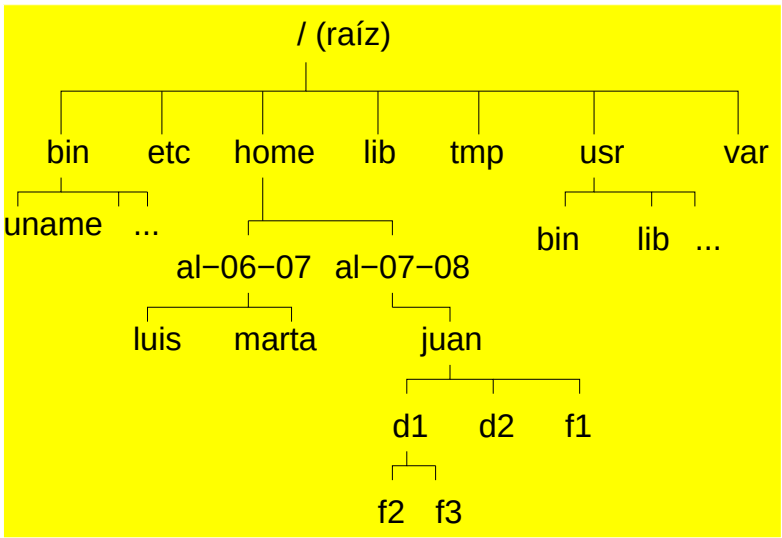
Muy Importante:
Comprueba los permisos de tu `HOME`, en muchos sistemas por omisión está abierto
Atención,
un fichero sin permisos de escritura, p.e. `rwxr-xr-x`
pero con permiso de escritura en el directorio que lo contiene, `rwxrwxrwx`
no podrá ser modificado pero sí borrado o renombrado

Directorios Especiales

- Todo directorio contiene dos subdirectorios especiales:

  `.` El subdirectorio `.` de un directorio es él mismo

  `..` El subdirectorio `..` de un directorio es su directorio padre



- Ejemplos:
  - El subdirectorio `.` de `al-07-08` es `al-07-08`
  - El subdirectorio `..` de `al-07-08` es `home`
  - El subdirectorio `..` de `home` es `/`

Variables

- `variable=valor`
  `echo $variable`
  Sin espacios antes y despues del igual
  con $ para acceder al contenido de la variable
  sin $ en la asignación
  sólo son visibles en ese proceso

```
nombre=juan
echo $nombre
```

Variables de entorno

- `export VARIABLE=valor`
  hace que los procesos hijos del proceso donde se declara la variable, la reciban. Por convenio se usan mayúsculas
- Para que el cambio sea permanente, hay que exportar la variable en algún fichero de configuración como p.e. `.bashrc`
- `printenv`
  muestra todas las variables de entorno
- HOME
- HOSTNAME
- USER
- PATH
  Contiene la lista de directorios donde la shell buscará los ejecutables (si no se indica path explícito)

La variable de entorno HOME

- Indica el directorio *hogar* de un usuario: el sitio donde se espera que cada usuario escriba sus cosas

```
koji@mazinger:~$ echo $HOME
/home/koji
```

- Se le suele llamar `$HOME`, pero esto no es muy preciso
  - La variable se llama `HOME`, el dólar se antepone a todas las variables en bash cuando se están referenciando (y no cuando se asignan)
  - Es un error frecuente intentar usar `$HOME` en otros lenguajes o en cualquier programa. Solo es válido en bash y shells similares

Virgulilla

La virgulilla (~) representa el directorio *home* de un usuario

- Equivale a `$HOME`, con la ventaja de que se puede usar en muchos lenguajes, aplicaciones y librerías (no todos)
- No aparece en los teclados, pero está accesible en `AltGr 4`
- Seguida de un nombre de usuario, representa el *HOME* de ese usuario

```
koji@mazinger:~$ echo ~jperez
/home/jperez
```

Si el nombre del usuario no es una cadena literal sino una variable es necesario volver a evaluar la expresión

```
koji@mazinger:~$ nombre=koji
koji@mazinger:~$ echo ~$nombre
~koji
koji@doublas:~$ eval echo ~$nombre
/home/koji
```

La variable de entorno PATH

Un usuario principiante ejecuta

```
koji@mazinger:~/pruebas$ ls -l
total 4
-rw-r--r-- 1 koji koji 27 2009-10-07 19:02 holamundo
```

Intenta invocar el mandato *holamundo* escribiendo

```
koji@mazinger:~/pruebas$ holamundo
```

pero obtiene

```
bash: holamundo: orden no encontrada
```

Problema 1

El fichero no tenía permisos de ejecución

Problema 1: Solución

```
koji@mazinger:~/pruebas$ chmod ugo+x holamundo
```

¿Problema resuelto?

```
koji@mazinger:~/pruebas$ ls -l
total 4
-rwxr-xr-x 1 koji koji 27 2009-10-07 19:02 holamundo
```

No ha bastado. El usuario vuelve a ejecutar

```
koji@mazinger:~/pruebas$ holamundo
```

pero vuelve a obtener

```
bash: holamundo: orden no encontrada
```

Problema 2

Aunque el fichero está en el directorio actual (directorio *punto*), la shell no lo buscará allí, sino donde indique la variable de entorno PATH, que contiene una lista de directorios, separados por el carácter *dos puntos*

```
koji@mazinger:~/pruebas$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Lo buscará en `/usr/local/sbin`
Si no lo encuentra, lo buscará en `/usr/local/bin`
Si sigue sin encontrarlo, lo buscará en `/usr/local/sbin`
etc
Pero no lo buscará en el directorio *punto*

Problema 2: Solución 1 (recomendada)

Invocar el mandato indicando explícitamente que el fichero está en el directorio *punto*

```
koji@mazinger:~/pruebas$ ./holamundo
¡hola mundo!
```

Problema 2: Solución 2

Indicar el trayecto absoluto del mandato

```
koji@mazinger:~/pruebas$ /home/koji/pruebas/holamundo
¡hola mundo!
```

Problema 2: Solución 3

Modificamos la variable de entorno PATH para añadir al final el directorio *punto*

Como queremos que el cambio sea permanente, debemos modificar la variable en un fichero de configuración [1] , por ejemplo `~/.bashrc`

```
export PATH=$PATH:.
```

El cambio no se produce de inmediato, sino cuando se ejecute de nuevo `~/.bashrc`

- Al invocarlo explícitamente

  ```
  koji@mazinger:~/pruebas$ source ~/.bashrc
  ```

- Al abrir una nueva terminal

[1] Más detalles en el apartado *invocación de la shell*

Problema 2: Solución 4 ¡Muy peligrosa!

Modificamos la variable de entorno PATH para añadir al principio el directorio *punto*

```
export PATH=.:$PATH
```

Supongamos que un atacante escribe un script con el nombre `ls` y el contenido

```
#!/bin/bash
rm -rf  $HOME
```

Al escribir la orden `ls`, se ejecutaría este script, y no `/bin/ls`

Directorio de Trabajo

- La shell en todo momento se encuentra en un cierto punto del árbol de ficheros. A ese punto se le llama **directorio de trabajo** (*working directory*)
- Normalmente la shell indica el directorio de trabajo en el *prompt*
- pwd: Muestra el directorio de trabajo actual (*print working directory*)

```
pwd
```

Trayectos (Paths)

- Un trayecto (path) consiste en escribir el camino hasta un fichero o directorio, incluyendo directorios intermedios separados por el carácter /
- Trayecto absoluto:
  - Escribe el camino desde el **directorio raíz**
  - **Siempre** empieza por /
- Trayecto relativo:
  - Escribe el camino desde el directorio de trabajo
  - **Nunca** empieza por /
- Cualquier programa acepta (o debería aceptar) que cuando se especifica un nombre de fichero, se use o bien la forma relativa o bien la forma absoluta.
  Esto es aplicable a casi cualquier programa de casi cualquier Sistema Operativo

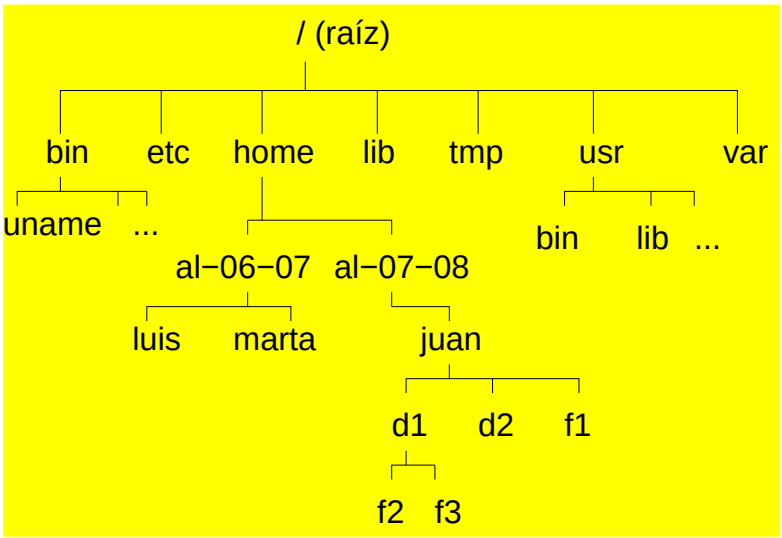
¿Un trayecto con virgulilla es relativo o absoluto?
`~/mi_directorio`
En cierta forma es relativo

- No empieza por /
- Depende del usuario que lo ejecuta

En cierta forma es absoluto

- No depende del directorio de trabajo
- Lo que sucede realmente es que se reemplaza la virgulilla por el trayecto absoluto del *home* del usuario

Posiblemente lo más adecuado es considerarlo un caso un poco especial de path absoluto



- Ejemplos:
  - Trayecto absoluto de `f2`:

    `/home/al-07-08/juan/d1/f2`
  - Trayecto relativo de `f2` si el directorio de trabajo es `juan`:

    `d1/f2`
  - Trayecto relativo de `f2` si el directorio de trabajo es `d2`:

    `../d1/f2`
  - Trayecto relativo de `var` si el directorio de trabajo es `luis`:

    `../../../var`

- cd: Cambia el directorio de trabajo (*change directory*)

`cd d1`	Cambia desde el directorio de trabajo actual a su subdirectorio `d1`
`cd /home`	Cambia desde cualquier directorio al directorio `/home`
`cd ..`	Cambia desde el directorio de trabajo actual a su directorio padre (sube un directorio)
`cd`	Cambia al directorio por defecto (hogar) del usuario

- ls: Muestra los contenidos de un directorio (*list*)

`ls`	Muestra el contenido del directorio de trabajo
`ls d1`	Muestra el contenido del subdirectorio `d1`
`ls /home`	Muestra el contenido de `/home`

touch

Cambia la fecha a un fichero, o lo crea si no existe

`touch <fichero>`

- Si `<fichero>` existe, le pone la fecha actual
- Si `<fichero>` no existe, crea un fichero vacío con este nombre

`touch -d <fecha/hora>  <fichero>`

Modifica la fecha de último acceso al fichero

```
touch -d 2007-02-28 fichero      # cambia la fecha
touch -d 15:41 fichero           # cambia la fecha
```

mkdir: Creación de directorios

mkdir: Crea directorios (*make directory*)

`mkdir <fichero>`

- `mkdir d3`
  Crea d3 como subdirectorio del directorio actual
- `mkdir d4 d5`
  Crea d4 y d5 como subdirectorios del directorio de trabajo actual
- `mkdir /tmp/ppp`
  Crea el directorio `/tmp/ppp`
- `mkdir -p d6/d7`
  Crea debajo de directorio de trabajo `d6` (si no existe), y crea `d7` debajo de `d6`

Copiar, mover y renombrar

- La órden cp copia ficheros
- La órden `mv` mueve y renombra ficheros

En primer lugar mostraremos el uso básico, después las opciones completas

Copiar un fichero:

tengo

```
/tmp/probando/quijote.txt
```

quiero

```
/tmp/probando/quijote.txt
/tmp/probando/quijote_repetido.txt
```

hago

```
cd /tmp/probando
cp quijote.txt quijote_repetido.txt
```

Renombrar un fichero:

tengo

```
/tmp/probando/quijote.txt
```

quiero

```
/tmp/probando/don_quijote.txt
```

hago

```
cd /tmp/probando
mv quijote.txt don_quijote.txt
```

Copiar un fichero en un directorio distinto

```
tengo
/tmp/probando/quijote.txt
```

quiero

```
/tmp/probando/quijote.txt
/tmp/otro_probando/quijote.txt
```

voy al directorio destino

```
cd /tmp/otro_probando/
```

```
#copio     "el fichero"          "aquí"
cp    /tmp/probando/quijote.txt    .
```

Mover un fichero a un directorio distinto

tengo

```
/tmp/probando/quijote.txt
```

quiero

```
/tmp/otro_probando/quijote.txt
```

voy al destino

```
cd /tmp/otro_probando/
```

```
# muevo "el fichero"          "aquí"
mv /tmp/probando/quijote.txt    .
```

cp: Copiar 1 fichero ordinario

`cp <origen> <destino>`

cp (*copy*) con dos argumentos. `<origen>` es un fichero ordinario

- Si el segundo argumento es un directorio
  Hace una copia del fichero `<origen>` dentro del directorio `<destino>`
- Si el segundo argumento NO es un directorio (es un fichero o no existe nada con ese nombre)
  Hace una copia del fichero `<origen>` y le pone como nombre `<destino>`

Como siempre, tanto `<origen>` como `<destino>` pueden indicarse con trayecto relativo o con trayecto absoluto

Ejemplos:

```
cp holamundo.py /tmp
cp ~/prueba.txt .
cp /home/jperez/prueba.txt prueba2.txt
```

cp: Copiar 1 directorio

`cp -r <origen> <destino>`
Si `<origen>` es un directorio, es necesario añadir la opción `-r` (*recursive*)

- Si `<destino>` es un fichero ordinario, se produce un error
- Si `<destino>` es un directorio, el directorio `<origen>` se copia dentro
- Si `<destino>` no existe, se le pone ese nombre a la copia

Ejemplos

```
cp -r ~  /tmp
cp -r /var/tmp/aa .
cp -r ~  /tmp/copia_de_mi_home
```

cp: Copiar varios ficheros ordinarios

`cp <origen1> <origen2> .... <destino>`
cp (*copy*) con varios argumentos. Los ficheros `<origen1> <origen2> ....` se copian en el directorio `<destino>`

- `<destino>` tiene que ser un directorio (o se producirá un error)
- `<origen1>, <origen2>,...` tienen que ser ficheros ordinarios (o un mensaje indicará que no se están copiando)

Ejemplos:

```
cp holamundo.py /home/jperez/prueba1.txt ../prueba2.txt /tmp
cp bin/*.py /tmp
```

cp: Copiar varios ficheros o directorios

`cp -r <origen1> <origen2> .... <destino>`
Este caso es idéntico al anterior, solo que si `<origen1>` o `<origen2>` o `...` son directorios, es necesaria la opción `-r`
Ejemplos:

```
cp -r holamundo.py /home/jperez /tmp
```

mv: mover o renombrar ficheros y directorios

`mv <origen> <destino>`

Mover dentro del mismo directorio equivale a renombrar

`<origen>` es un fichero o un directorio

- Si el segundo argumento es un directorio
  Mueve `<origen>` dentro del directorio `<destino>`
- Si el segundo argumento no existe
  Mueve `<origen>` a `<destino>`
- Si `<destino>` es un fichero
  - y `<origen>` es un fichero, `<origen>` pasa a llamarse `<destino>` y el anterior `<destino>` despararece
  - y el primero es un directorio, se produce un error

Ejemplos:

```
mv holamundo.py /tmp
mv ~/prueba.txt .
mv /home/jperez/prueba.txt prueba2.txt
```

mv con más de dos argumentos

`mv <origen1> <origen2> ... <destino>`

`<destino>` debe ser un directorio existente

`<origen1>, <origen2>...` pueden ser ficheros ordinarios o directorios

Ejemplos:

```
mv holamundo.py /home/jperez/prueba1.txt ../prueba2.txt /tmp
mv *.txt texto
```

Tipos de fichero

- Tradicionalmente en Unix los ficheros no llevaban extensión
  No hay un programa asociado a cada extension
  ```
  file mifichero
  ```
  Indica el tipo del fichero. No importa si tiene extensión, si no la tiene, o si es errónea

Supongamos que tenemos un fichero y no sabemos con qué programa podemos abrirlo. P.e. desconocemos que tenemos instalado evince para abrir ficheros pdf

- En Linux
  - Si nuestro escritorio es gnome, podemos ejecutar
    `gnome-open fichero.extension`
  - Si usamos KDE, `kde-open fichero.extension`
  - Para gnome, KDE y muchos otros
    `xdg-open fichero.extension`
- En Mac OS
  `open fichero.extension`

Borrado de un fichero

- `rm` *fichero*
  borra fichero [2]
  `rm -r` *directorio*
  Borra un directorio y todo su contenido

[2] Cuando hablemos de enlaces veremos una definición más exacta

Un usuario de MS-DOS podría intentar hacer

```
mv *.txt *.doc   # ¡MAL!  No funciona, y puede ser fatal
```

Supongamos que tenemos en el directorio actual

```
carta1.txt
carta2.doc
```

Tras expandir los asteriscos, el resultado es

```
mv carta1.txt carta2.doc  # destruimos el segundo fichero!
```

Una solución posible [3]:

```
#!/bin/bash
for fichero in *.txt
do
    nombre=$(echo $fichero | cut -d. -f1)
    extension=$(echo $fichero | cut -d. -f2)
    mv $fichero $nombre.doc
done
```

[3] Siempre que solo haya un punto en el nombre

Enlace duro

Un nuevo nombre para el fichero
`ln a b`

- Ambos nombres deben pertenecer al mismo sistema de ficheros
- Dado un fichero, se sabe cuántos nombres tiene. Para saber cuáles son sus nombres, habría que buscarlos
- La mayoría de los S.O. no permiten enlaces duros a directorios, puesto que podría provocar bucles difíciles de detectar

`rm` borra un nombre de un fichero
si es el último, borra el fichero.

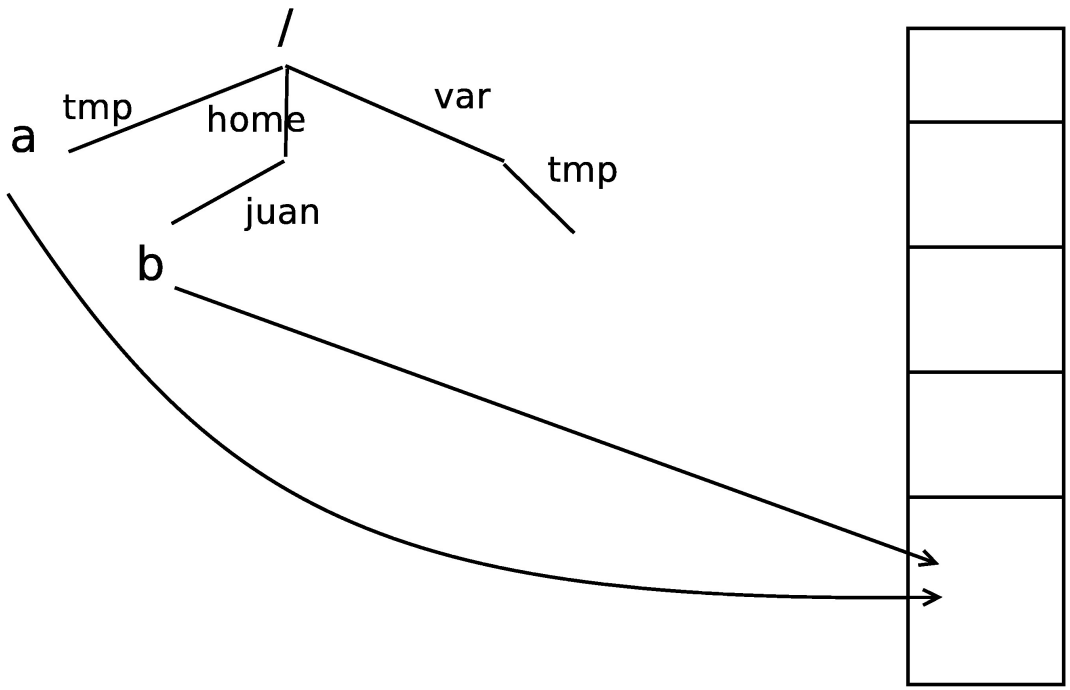


Figura: Enlace Duro

Enlace blando o simbólico

Un nuevo fichero que apunta a un nombre
`ln -s /home/juan/b c`

- Sirven principalmente para mantener ficheros ordenados y *a mano*
- Puede hacerse entre distintos sistemas de ficheros
- Puede enlazarse un directorio
- Con enlaces simbólicos, si se borra el original
  el enlace queda roto
- El fichero original podemos especificarlo
  - Con su path absoluto
  - Con su path relativo
    En este caso, si movemos el enlace simbólico pero no movemos el original, se pierde la referencia

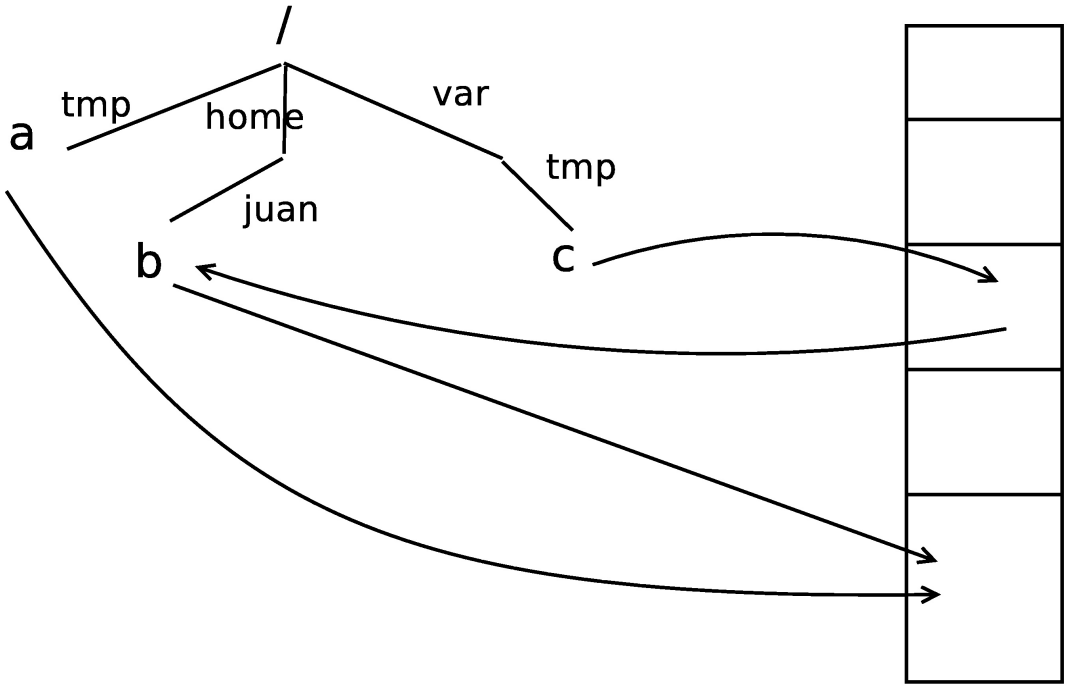


Figura: Enlace Simbólico

Utilidad de los enlaces

Tanto los *blandos* como los *duros*
son útiles:

- Para tener acceso a un fichero en un trayecto más *cómodo*, más *a mano*
- Si cambio de criterio sobre el lugar o el nombre de un fichero. Mediante un enlace, el fichero sigue accesible tanto por el nombre antiguo como por el nuevo

Ventaja de los enlaces duros:

- Protegen frente a borrados accidentales de un nombre. Pero no frente a ningún otro problema que pueda tener el fichero, por tanto su utilidad es mínima

Ventaja de los enlaces simbólicos:

- Se pueden establecer entre sistema de ficheros distintos
- Se pueden usar para directorios

Los enlaces simbólicos se usan mucho más que los enlaces duros

Mandatos de uso básico de la red

ping: Comprueba si una máquina responde en la red

`ping gsyc.es`	Sondea la máquina `gsyc.es` indefinidamente mostrando el doble de la latencia con ella. `CTRL-c` para terminar y mostrar un resumen
`ping -c 4 gsyc.es`	Sondea la máquina `gsyc.es` 4 veces

traceroute: Muestra encaminadores intermedios hasta un destino

`traceroute gsyc.es`	Muestra encaminadores intermedios desde la máquina en la que se está hasta `gsyc.es`. Muestra el doble de las latencias hasta cada punto intermedio.

```
traceroute to gsyc (193.147.71.64), 30 hops max, 60 byte packets
 1  ap (192.168.1.1)  0.730 ms  1.376 ms  1.345 ms
 2  10.213.0.1 (10.213.0.1)  9.927 ms  15.040 ms  15.029 ms
 3  10.127.46.153 (10.127.46.153)  15.003 ms  15.632 ms  15.607 ms
 4  mad-b1-link.telia.net (213.248.90.85)  28.549 ms  28.720 ms  28.691 ms
 5  dante-ic-125710-mad-b1.c.telia.net (213.248.81.26)  28.822 ms  28.959 ms  3
 6  nac.xe0-1-0.eb-madrid0.red.rediris.es (130.206.250.22)  36.344 ms  35.077 m
 7  cam-router.red.rediris.es (130.206.215.66)  34.940 ms  12.015 ms  12.689 ms
 8  * * *
 9  gsyc.escet.urjc.es (193.147.71.64)  14.675 ms  14.934 ms  15.500 ms
```

ssh

Ejecuta mandatos de shell en una máquina remota

`ssh jperez@zeta12.pantuflo.es`
: Se conecta a la máquina `zeta12.pantuflo.es` (pide password) y permite ejecutar mandatos en ella.
Toda la sesión entre la máquina origen y destino viaja cifrada por la red

`ssh jperez@zeta12.pantuflo.es ls /`
: Se conecta a la máquina `zeta12.pantuflo.es` (pide login y password), ejecuta el mandato `ls /` y sale de ella.

- La primera vez que abrimos una sesión en una máquina, ssh nos indica la huella digital de la máquina remota

```
The authenticity of host 'gamma23 (212.128.4.133)' can't be established.
RSA key fingerprint is de:fa:e1:02:dc:12:8d:ab:a8:79:8e:8f:c9:7d:99:eb.
Are you sure you want to continue connecting (yes/no)?
```

- Si necesitamos la certeza absoluta de que esta máquina es quien dice ser, deberíamos comprobar esta huella digital por un medio seguro, alternativo
- La sesión se cierra cerrando la shell remota (`exit` o `ctrl d`)

scp

`scp [[loginname@]maquina:]<origen> [[loginname@]maquina:]<destino>`

Copia ficheros desde/hacia máquinas remotas. El contenido de los ficheros viaja cifrado por la red.
Igual que cp, pero ahora hay que añadir o bien a `origen` o bien a `destino`

- ¿Cuál es la máquina remota?
- ¿Qué nombre de usuario tenemos en la máquina remota?

  `usuario@maquina:`

- En caso de que el nombre de usuario en la máquina local sea el mismo que en la máquina remota, puede omitirse `usuario@`
- Los dos puntos del final nunca pueden omitirse
- No puede haber espacios después de los dos puntos
- La máquina se puede indicar por su nombre o por su dirección IP
- Naturalmente, origen y destino pueden indicarse con trayecto relativo o con trayecto absoluto
  - En la máquina remota, los trayectos relativos parten del *home* del usuario remoto

Ejemplos:

`scp f1 jperez@alpha.aulas.gsyc.urjc.es:d1/f1`
: Lleva una copia del fichero `f1` desde la máquina local hasta la máquina `alpha`, entrando como usuario `jperez`, con trayecto `~jperez/d1/f1`

`scp f1 jperez@alpha.aulas.gsyc.urjc.es:`
: Lleva una copia del fichero `f1` desde la máquina local hasta la máquina `alpha` , entrando como usuario `jperez`, con trayecto `~jperez/f1`

`scp jperez@alpha.aulas.gsyc.urjc.es:f1 .`
: Trae desde la máquina `alpha`, entrando con el usuario `jperez`, el fichero `~jperez/f1` hasta el directorio de trabajo de la máquina local

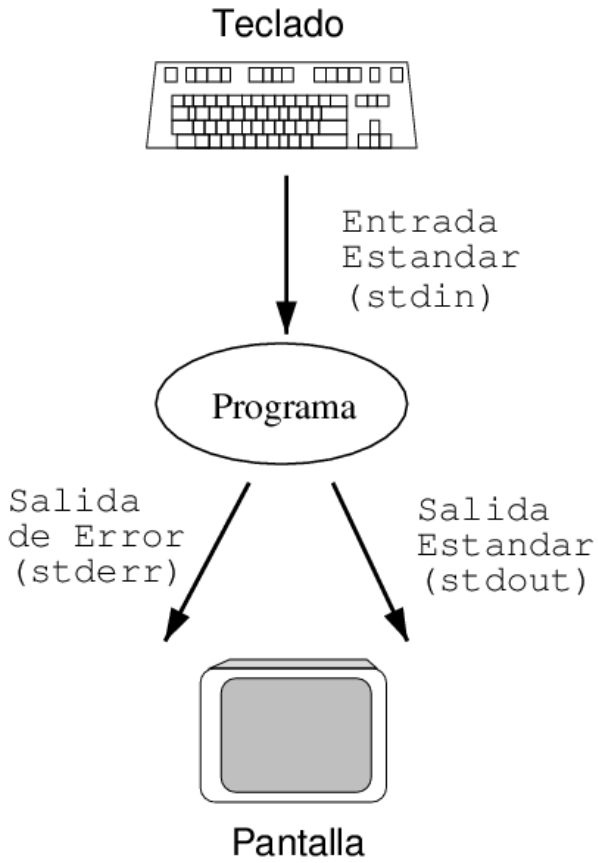
Recuerda:

`~jperez`	*home* de jperez
`~/dir1`	subdirectorio `dir1` dentro de mi *home*

Si scp resulta nuevo para tí y no quieres equivocarte, puedes seguir estos pasos:

1. Ten dos sesiones abiertas, una la máquina origen y otra en la máquina destino
2. Mediante `cd`, vete al directorio origen en la máquina origen y haz `pwd` para asegurarte de que estás donde debes
3. Mediante `cd`, vete al directorio destino en la máquina destino y haz `pwd` para asegurarte de que estás donde debes
4. En la máquina origen, haz `ls` del fichero, indicando el path de forma absoluta. El `pwd` anterior te ayudará. Si te equivocas, te darás cuenta ahora
   ```
   ls /path/absoluto/al/fichero.txt
   ```
5. Ejecuta el `scp` en la máquina destino. Especifica el origen con la ayuda de un copia-y-pega del paso anterior. Especifica el destino con '.'
   ```
   scp usuario@maquina:/path/absoluto/al/fichero.txt .
   ```

Entrada y salida



- entrada estándar
- salida estándar
- salida de error estándar

Paso de argumentos a órdenes

Muchas órdenes se comportan así (no todas)

- Sin argumentos: Entrada estándar

  ```
  wc
  ```

- 1 argumento: Nombre de fichero

  ```
  wc fichero
  ```

- n nombres de fichero

  ```
  wc fichero1 fichero2
  ```

- `cat`
  lee lo que hay en stdin y lo escribe en stdout
  (Ctrl D: fin de fichero)
- `cat fichero1 fichero2`
  lee los ficheros que se pasan como argumento y los escribe (concatenados) en stdout
  (Ctrl D: fin de fichero)
- `echo argumento`
  escribe en stdout el texto que se le pasa como argumento.
  Añade retorno de carro
- `echo -n argumento`
  escribe en stdout el texto que se le pasa como argumento
- `less fichero`
  escribe un fichero en stdout, permitiendo paginación

Redirecciones

```
<    redirige stdin desde fichero
>    redirige stdout a fichero, reemplazando
>>   redirige stdout a fichero, añadiendo
|    redirige stdout de un proceso a stdin del siguiente
```

- `cat`
- `cat file1 file2 > file3`
  `cat file1 | less`
  `cat > file1`
- `less fichero`
  `cat fichero | less`
  `less < fichero`
  (El resultado es el mismo, pero es importante distinguirlo)

1 representa stdout

2 representa stderr

- `mkdir /a/b/c 2> mi_fichero_errores`
  Redirige stderr al fichero
- `cp fichero_a fichero_b 2>/dev/null`
  Redirige stderr al fichero *sumidero* (Lo que se copia en `/dev/null` desaparece sin mostrarse)

Para escribir en 1 o en 2, es necesario anteponer `&` (para que no se confunda con un fichero que se llame `"1"` o `"2"`)

- `echo "ERROR: xxxx ha fallado"  >&2`
  Redirige el mensaje a stderr

`&` representa stdout y stderr

- `find /var  &>mi_fichero`

sudo y redirecciones

La orden *sudo* por omisión no incluye las posibles redirecciones

- `sudo echo hola > /tmp/aa`
  El proceso *echo* se lanza con la identidad del root (id 0), pero la redirección la ejecuta el usuario ordinario
- Para poder usar redirecciones, ejecutamos una subshell con el parámetro -c

```
sudo bash -c  "echo hola>aa"
sudo bash -c  "find /root | grep prueba "
```

Programación de Scripts

- En esta asignatura generalmente programaremos los scripts en python, que es más potente y sencillo que bash
- Pero para tareas muy básicas (cp, mv, ln -s, etc) puede ser más conveniente un script de bash

```
#!/bin/bash
a="hola mundo"
echo $a
```

Para invocarlo:

```
koji@mazinger:~$ ./holamundo
hola mundo
```

Es recomendable que un script empiece por `#!/bin/bash`, pero no es imprescindible

```
a="hola mundo"
echo $a
```

En este caso podemos ejecutar una shell y pasarle como primer argumento el fichero

```
koji@mazinger:~$ bash holamundo
hola mundo
```

o bien ejecutar una shell y redirigir el fichero a su entrada estándar

```
koji@mazinger:~$ bash <holamundo
hola mundo
```

Esto también puede ser útil para ejecutar un script sin permiso de ejecución (basta el de lectura)

Filtros

- Los filtros son muy importantes en el scripting Unix: grep, sed, sort, uniq, head, tail, paste...
- Un mandato genera una salida, un filtro procesa la salida (selecciona filas o columnas, pega, reemplaza, cuenta, ordena...) y lo pasa al siguiente mandato
- Ejemplo

```
who | cut -c1-8 |sort |uniq | wc -l

ps -ef | grep miguel | grep -v gvim
```

- En esta asignatura programaremos en python (de nivel más alto y más intuitivo), así que solo usaremos filtros muy básicos

grep

- grep es un filtro que selecciona las filas que contengan (o que no contengan) cierto patrón
- Para definir patrones de texto, emplea expresiones regulares (regexp)
  - Las regexp de grep, sed y awk son *clásicas*.
  - Las regexp de perl, python y ruby son una evolución de las regexp clásicas. Son mucho más intuitivas
  - Para tareas muy sencillas, podemos usar grep o sed. Si nuestras necesidades son más complejas y podemos elegir qué herramienta usar, mejor python (o ruby)

grep con un argumento

- `grep <patrón>`
  Lee stdin y escribe en stdout las líneas que encajen en el patrón
- `grep -v <patrón>`
  Lee stdin y escribe en stdout las líneas que **no** encajen en el patrón
- `grep -i <patrón>`
  Lee stdin y escribe en stdout las líneas que encajen en el patrón, ignorando mayúsculas/minúsculas

Ejemplos

```
ps -ef | grep -i ejemplo
ps -ef | grep -v jperez
dmesg | grep eth
```

grep con dos o más argumentos

- `grep <patrón> <fichero_1> ... <fichero_n>`
  Lee los ficheros indicados y escribe en stdout las líneas que encajen en el patrón

Ejemplos

```
grep linux *.txt
grep -i hidalgo quijote.txt
grep -v 193.147 /etc/hosts
```

Atención: Si el patrón a buscar incluye espacios, es necesario escribirlo entre comillas.

- `grep "la mancha" quijote.txt`
  Busca el patrón *la mancha* en el fichero *quijote.txt*
- `grep la mancha quijote.txt`
  Busca el patrón *la* en el fichero *mancha* y en el fichero *quijote.txt*

Atención:

- Hablamos de patrones, no de palabras. El patrón *ana* encaja en la palabra *ana* pero también en *rosana*
- Los metacaracteres de las regexp no son iguales que los metacaracteres (comodines) del bash

Algunos metacaracteres:

- `grep -i '\<ana\>'`
  Principio de palabra, patrón *ana*, final de palabra. Insensible a mayúsculas. (Dicho de otro modo, la palabra *ana*, sin confusión con *Mariana*)
- `grep -i '\<ana p.rez\>'`
  El punto representa cualquier carácter (equivalente a la interrogación en las shell de bash)
- `grep -i '\<ana p[eé]rez\>'`
  Después de la *p* puede haber una *e* con tilde o sin tilde

xargs

Mediante pipes podemos formar filtros concatenando órdenes. Pero ¿qué sucede cuando la información la necesitamos como parámetro, no en la entrada estándar?

```
locate -i basura | rm    # ¡Esto NO FUNCIONA!
```

Podemos usar la orden xargs

```
locate -i basura | xargs rm
```

Ejecuta rm tantas veces como líneas haya en stdin. Y le pasa cada línea como argumento

- Cuando necesitamos que la línea de entrada vaya en una posición distinta, usamos la opción `-I replstr` , donde `replstr` es la *replace string*, la cadena que reemplazaremos por el argumento
- El valor recomendado es `{}`, porque no es fácil que aparezca en otro sitio

```
locate  basura | xargs -I {} mv {} /tmp/papelera
find . | grep -i jpg | xargs -I {} mv {} /tmp/fotos
```