Tema 5: Uso del shell

Sistemas Operativos II

Tomás Fdez. Pena

tf.penas@usc.es

Contents

1. Acceso a la documentación

Necesitamos acceder a información sobre los diferentes comandos y opciones de UNIX

- Además de Internet, en el propio sistema existe información esencial:
  1. En las páginas de manual: comando `man`
  2. En las páginas de información: comando `info`
  3. Documentación de las aplicaciones en `/usr/share/doc`

1.1. Páginas de manual

Mejor lugar para buscar información concreta sobre la sintaxis y opciones de comandos y utilidades
Acceso a través del comando `man`

```
$ man ls
```

La salida de `man` es como sigue:

1. cabecera: nombre y sección del manual

```
LS(1)                    User Commands                    LS(1)
```

2. Nombre y descripción corta del comando

```
NAME
       ls - list directory contents
```

3. Sintaxis del comando

```
SYNOPSIS
       ls [OPTION]... [FILE]...
```

4. Descripción y opciones

```
DESCRIPTION
       List information about the FILEs (the current directory
       by default).  Sort entries alphabetically if none of
       -cftuSUX nor --sort.

       Mandatory arguments to long options are mandatory for
       short options too.

       -a, --all
              do not hide entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              print the author of each file

       -b, --escape
              print octal escapes for nongraphic characters
```

5. Otra información:

```
AUTHOR
       Written by Richard Stallman and David MacKenzie.

REPORTING BUGS
       Report bugs to <bug-coreutils@gnu.org>.

COPYRIGHT
       Copyright © 2004 Free Software Foundation, Inc.
       This  is  free  software;  see  the  source for copying
       conditions.  There is NO warranty; not even for MER-
       CHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

SEE ALSO
```

```
The full documentation for ls is maintained as a Texinfo manual.
If the info and ls programs  are  properly
installed at your site, the command

       info coreutils ls

should give you access to the complete manual.
```

Para navegar a través del manual usar las flechas, y la barra espaciadora

- para buscar un texto usar `/`*`texto`*, por ejemplo: `/option`
  - para repetir la búsqueda: `n`
- La tecla `Q` sale del manual

Organización de las páginas de manual

Las páginas de manual se organizan en secciones:

Sección	Uso
1	Comandos de usuario y aplicaciones
2	Llamadas del sistema
3	Llamadas de la biblioteca
4	Ficheros especiales (se encuentran generalmente en `/dev`)
5	Formato de ficheros y convenciones (p.e. `/etc/passwd`)
6	Juegos
7	Ficheros varios, incluyendo macros
8	Comandos de administración del sistema (generalmente sólo para root)
9	Rutinas del núcleo (no estándar)

`man` busca en las diferentes secciones siguiendo el orden 1,8,2,3,4,5,6,7,9

- si un comando aparece en varias secciones, se muestra la primera que encuentra
- para ver todas las páginas de manual de un comando usar `man -a`
  - muestra una página detrás de otra
- para buscar en una sección concreta usar `man` *`nseccion`*

```
$ man 5 passwd
```

Las páginas para cada sección se almacenan en un directorio diferente, normalmente `/usr/man/man`*x* o `/usr/share/man/man`*x*, con *x* el número de sección

- Podemos saber la localización de la página de un comando con `man -w`

  ```
  $ man -w passwd
  /usr/share/man/man1/passwd.1.gz
  ```

- Para ver todos los ficheros `man -wa`

  ```
  $ man -wa passwd
  /usr/share/man/man1/passwd.1.gz
  /usr/share/man/man1/passwd.1ssl.gz
  /usr/share/man/man5/passwd.5.gz
  ```

- Puede haber páginas de manual localizadas en otros directorios:
  - p.e., aplicaciones externas en `/usr/local/man`
  - los directorios donde buscar páginas de manual se indican en la variable de entorno `MANPATH` o en el fichero `/etc/manpath.config`
    - ver el comando `manpath` para más información

Comando `whatis`

El comando `whatis` o `man -f`, dado un comando nos muestra una descripción breve del mismo, tal como aparece en las páginas del manual

```
$ whatis passwd
passwd (1)           - change user password
passwd (5)           - The password file
passwd (1ssl)        - compute password hashes
```

Esta búsqueda la obtiene de una base de datos de `whatis`

- esa base se construye con los comandos `makewhatis`, `catman`, o `mandb`, dependiendo de la distribución

Comando `apropos`

El comando `apropos` o `man -k` busca en la base de `whatis` por el nombre del comando y su descripción

```
$ apropos passwd
chpasswd (8)         - update password file in batch
dpasswd (8)          - change dialup password
fgetpwent_r (3)      - get passwd file entry reentrantly
getpwent_r (3)       - get passwd file entry reentrantly
gpasswd (1)          - administer the /etc/group file
kdepasswd (1)        - graphical frontend to change the user's password
ldappasswd (1)       - change the password of an LDAP entry
lppasswd (1)         - add, change, or delete digest passwords.
mkpasswd (1)         - Overfeatured front end to crypt(3)
passwd (1)           - change user password
passwd (1ssl)        - compute password hashes
passwd (5)           - The password file
passwd2des (3)       - RFS password encryption
smbpasswd (8)        - change a user's SMB password
update-passwd (8)    - safely update /etc/passwd, /etc/shadow and /etc/grou
```

1.2. Otra documentación

Además de las páginas de manual existen otras fuentes de información en el sistema

Info

Sistema de manuales del proyecto GNU

- más flexible y completo que `man`
- proporciona información basada en hipertexto
  - la información se organiza en nodos en forma de árbol
  - se puede navegar por los nodos y a través de menús
- la información se lee a través del comando `info` o del programa `emacs`
- Ejemplo: `info coreutils`

Información sobre aplicaciones

En el directorio `/usr/doc` o `/usr/share/doc` se guarda información sobre las aplicaciones:

- Ficheros README y FAQ (*Frequently Asked Questions)*
- Documentación en HTML
- HOWTOs

2. El editor `vi`

Editor estándar en UNIX

- Orientado a pantalla (a diferencia de `ed` o `ex` que son orientados a línea)
- Disponible en todos los UNIX y muchos otros S.O. (incluídos MS-DOS y Windows)
- Funciona en una gran variedad de sistemas y terminales
- Esencial para un administrador de sistemas

Existen otros editores: `emacs`, `XEmacs`, `pico` (o `nano`), `joe`,... (lista de editores, comparativa y guerra)

- elegir el que más nos apetezca, pero
  - `vi` es uno de los más potentes,
  - es pequeño y rápido,
  - no es tan complicado como parece y,
  - está siempre disponible

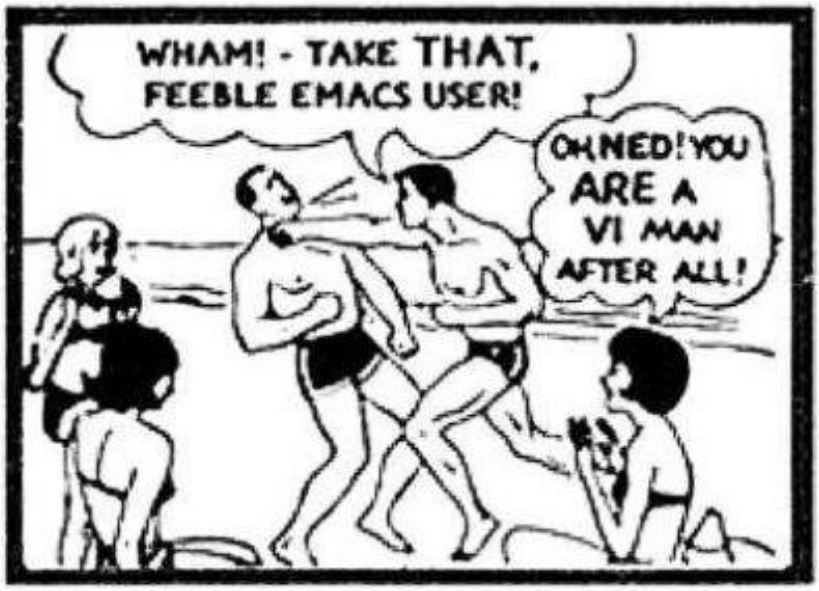
Existen variantes y mejoras del `vi` original: `vim`, `gvim`, `elvis`, `cream`,... que pueden facilitar el uso

- algunas necesitan un entorno de ventanas (p.e. `gvim` o `cream`)
- `vim` es el más usual en Linux

Páginas sobre `vi`

- The Vi Lovers Home Page
- Tutorial de Vim en castellano
- vi intro - the cheat sheet method
- Carta de referencia rápida
- Vi graphical cheat sheet and tuturial
- Fanáticos: The Cult of vi
- La oposición: The Church of Emacs
- Ed, man! !man ed: Ed is the standard text editor

La realidad



Inicio de vi
Para iniciar `vi` con un fichero vacío, simplemente escribir `vi`
Otras opciones:

Línea de comandos	
`vi` *fichero*	Editar un fichero (si el fichero es de solo lectura lo editara en modo solo lectura)
`vi` *f1 f2 f3*	Editar varios ficheros sucesivamente
`vi -R` *fichero*	Editar en modo solo lectura
`vi -r` *fichero*	Editar recuperando modificaciones no salvadas
`vi +`*n fichero*	Comienza a editar el fichero en la línea *n*
`vi +/`*string fichero*	Abre el fichero, y salta a la línea con la primera aparición de *string*
`vi -h`	Más opciones

2.1. Modos de funcionamiento

`vi` tiene tres modos de funcionamiento:

1. Modo comandos

   Permite mover el cursor, introducir comandos, borrar, copiar y pegar, y entrar en el modo inserción

2. Modo inserción

Permite insertar y borrar texto (dependiendo de la configuración podremos movernos o no a través del texto con las flechas)
Vuelve a modo comandos con `ESC`

3. Modo *ex* o última línea

   Permite usar comandos complejos; se entra en el modo desde el modo comandos con `:`

Modo comandos

Comandos de movimiento	
`h, j, k, l`	izq., abajo, arriba, derecha
`+, -`	primer carácter de la línea siguiente o anterior
`{, }`	salta párrafos
`0, $`	comienzo, final de la línea
`gg, G`	comienzo, final del fichero
n`G`, `:`*n*	ve a la línea n
`w, W, b, B`	adelante, atrás una palabra (may. palabra delimitada por blancos)
`H, M, L`	arriba, medio o abajo de la pantalla
`C-F, C-B`	adelante, atrás una página
`C-D, C-U`	adelante, atrás media página
Entrada de texto	
`i, a`	insertar/añadir antes/después del cursor
`I, A`	insertar/añadir al inicio/final de la línea
`o, O`	insertar una línea después/antes de la actual
`r, R`	reemplazar uno/múltiples caracteres
Marcar líneas	
`m`*b*	pon la marca *b* en la línea
`'`*b*	ve a la marca b

Copiar, Mover, y borrar lineas	
`x, X`	borra un carácter (adelante/atrás)
`dd`	borra una línea
`D`, `d$`	borra hasta el final de la línea
`yy, Y`	copia una linea en el buffer
`p, P`	pega el buffer después/antes del cursor
Corregir	
`u`	deshace último cambio (`vim` permite deshacer varios)
`U`	recupera línea completa
`C-R`	en `vim`: rehacer
Buscar	
`/`*patrón*	busca *patrón* hacia adelante
`?`*patrón*	busca *patrón* hacia atrás
`n, N`	siguiente/anterior
`%`	busca paréntesis
Varios	
`J`	unir dos líneas
`ZZ`	salvar y salir
`.`	repetir el último comando
`Ctrl-L`	refrescar pantalla
`Ctrl-G`	muestra información
`~`	cambia may./min.
`>>`	tab. al principio de línea

Buffers: `vi` admite 26 buffers con nombre, desde el `a` al `z`

- para diferenciar entre un comando y un buffer se usa `"` antes del nombre del buffer
  - `"byy` copia una linea en el buffer `b`
  - `"bp` pega el buffer `b`

Combinación de comandos: es posible combinar comandos, por ejemplo:

Combinación	**Acción**
`5j`	abajo 5 líneas
`db, dw`	borra del cursor al principio/final de la palabra
`d0, d$`	borra del cursor al principio/final de la línea
n`dd`	borra las siguientes n líneas
n`dw`	borra las siguientes n palabras
`d5G`	borra hasta la línea 5
`dG`	borra hasta el final del fichero
`y0, y$`	copia del cursor al principio/final de la línea
`yb, yw`	copia del cursor al principio/final de la palabra
n`yy`, *n*`Y`	copia las siguientes n líneas
n`p`	pega n copias del buffer
`"m3P`	pega 3 copias del buffer `m` antes del cursor
`"b5dd`	borra 5 líneas y mételas en el buffer `b`

Modo ex

Ficheros	
`:w`, `:w!`	Salvar (con ! fuerza sobreescribir)
`:r` *fich*	lee un fichero en la posición actual
`:n` `:prev`	fichero siguiente/anterior
`:e` *fich*	edita de forma simultanea otro fichero
`:e #`	vuelve al fichero anterior
`:q`, `:q!`	sale (con ! descarta cambios)
`:x`, `:wq`	sale y guarda cambios
Buscar y reemplazar	
`:s/`*s1*`/`*s2*`/`	Sustituir, en la línea actual, la primera ocurrencia de la cadena *s1* por *s2*
`:s/`*s1*`/`*s2*`/g`	Igual pero reemplaza todas las ocurrencias
`:s/`*s1*`/`*s2*`/gc`	Igual con confirmación

Abreviaturas y macros	
`:ab` *s1 s2*	Abreviaturas: teclear *s1* se sustituye por *s2*
`:unab` *s1*	Desabreviar
`:map` *m n*	Crear macro (m hace n)
`:unmap` *m*	Destruir macro
Comandos externos	
`:!` *c*	ejecuta el comando *c* en una subshell
`:sh`	inicia una subshell
Varios	
`:set nu`	muestra números de línea
`:set all`	ver todas las opciones con sus valores
`:syntax enable`	activa el *syntax highlighting*
`:split`, `:vsplit`	divide pantalla (`Ctrl-w`+flechas para cambiar)
`:help`	ayuda

Rangos Muchos comandos pueden aplicarse dentro de un rango de líneas especificado:

- Un `.` representa la línea actual
- Un `$` representa la última línea
- Un `%` representa todo el fichero (abreviatura de `1,$`)
- Ejemplos
  - `:5,10 w fich` salva en `fich` desde la línea 5 a la 10
  - `:.,+3 d` borra desde la línea actual hasta la actual +2
  - `:15,$s/hola/adiós/g` reemplaza desde la línea 15 al final todas las apariciones de `hola` por `adiós`
  - `:'c,'d co 'y` copia el texto desde la marca `c` hasta la `d` en `y`
  - `:'c,'d mo 'y` mueve el texto desde la marca `c` hasta la `d` en `y`

  - `:%s/$/^M/g` pone a doble línea todo el texto[1]

Macros Es posible definir macros para hacer tareas repetitivas

- Macro que mueve 4 líneas al final del fichero
  - `:map xxx 4ddGp`
    Al teclear `xxx` borramos 4 líneas (`4dd`), nos vamos al final del fichero (G), y pegamos (`p`) las cuatro líneas
  - `:map <F5> :w<CR>:!gcc %<CR>:!a.out<CR>`
    Asigna a `<F5>` la acción de compilar y ejecutar
- Con `:map` vemos las macros definidas
- Con `unmap` destruimos una macro

En Vim se pueden grabar macros:

- Para grabar (modo comandos): `q<`*`letra`*`><`*`comandos`*`>q`
- Para ejecutar (modo comandos): `<`*`número`*`>@<`*`letra`*`>`

Ver vim.wikia.com/wiki/Macros

Modo visual

Permite marcar de forma cómoda un trozo de texto para copiar y pegar

- Para pasar al modo visual desde el modo comandos: `v` (simple), `V` (por líneas) o `Ctrl-V` (por bloques)
- El texto se marca con las teclas de desplazamiento
- Copiamos y pegamos con las teclas usuales (`y` copiar, `d` cortar, `p` pegar)

El modo visual sólo está disponible en versiones modernas de `vi` (`vim`, `elvis`, etc.)

[1] Para introducir un carácter de control, como `^M`, tenemos que pulsar primero `Ctrl-V` y luego el carácter, en este caso `Enter`

2.2. Ficheros de configuración

Se pueden configurar y personalizar el `vi` a través de diversos ficheros (ver el manual del `vi` para más detalles)
En concreto, para `vim` podemos tener

1. Configuración global: `/usr/share/vim/vimrc`
2. Configuración por usuario: `~/.vimrc`

En este fichero de configuración podemos definir opciones particulares, macros, abreviaturas, etc.

2.3. vimtutor

La mejor (y única) forma de aprender vi es con la práctica

- Existen múltiples tutoriales on-line
- Una alternativa es usar el `vimtutor`
  - ejecutar `vimtutor` (`vimtutor -g es` para versión en castellano)

2.4. Prácticas: El editor vi

1. Ejecutar el `vimtutor` y seguir los pasos en el indicados
2. Usando el documento del `vimtutor` o otro cualquiera probar los comandos y macros antes comentados
   - uso del modo visual
   - establecimiento de marcas
   - uso de abreviaturas
3. Comprueba la utilidad de grabar una macro con el comando `q` y de reproducirla con el comando `@`. Para ello haz una macro que haga algo que no sea habitual encontrar en un editor de texto, por ejemplo, borrar las líneas que estén en posiciones múltiplo de 5, o mover las líneas que sean múltiplo de 4 dos posiciones más arriba añadiéndoles el carácter '.' al final, o borrar en las líneas en posiciones pares los caracteres desde el tercero en adelante, o sustituir el carácter 's' por 't' que sean inicio de palabra en las líneas pares...

3. La línea de comandos

Veremos conceptos básicos para usar nuestro sistema desde la línea de comandos:

1. introducción al interprete de comandos (*shell*)
2. uso de la linea de comandos
3. el histórico de comandos
4. sustitución de nombres de ficheros
5. redirección de la entrada/salida
6. variables de shell
7. sustitución de comandos
8. orden de evaluación
9. otras expansiones
10. definición de alias
11. ficheros de inicialización

3.1. El interprete de comandos (shell)

El shell nos proporciona:

- un interprete de comandos
- un entorno de programación

El shell nos permite ejecutar:

- Comandos externos, por ejemplo: `ls`, `cat`, `mkdir`, etc.
  - cuando se lanzan el shell crea un nuevo proceso (con `fork`) y cambia su imagen por la de dicho archivo ejecutable (con `exec`)
    - Cuando el shell crea ese nuevo proceso, se bloquea y espera a que acabe (con `wait`)
    - Este comportamiento se puede cambiar si se ejecuta el comando en segundo plano (background)

  - El shell mantiene una lista de directorios (ruta de búsqueda) en la que buscar esos ejecutables
    * Esta lista está especificada en la variable PATH

- Comandos internos (*builtin commands*), por ejemplo: `cd`, `bg`, `alias`, `eval`, `exec`, `pwd`, etc.
  - se ejecutan en el mismo proceso del shell, sin lanzar un nuevo proceso
  - ver el manual del shell para más información (o para el shell bash: `man bash-builtins`, o el comando `help`)
- En bash: para saber si un comando es externo o interno usar el comando interno `type`:

```
$ type cd
cd is a shell builtin
$ type cat
cat is /bin/cat
```

Principales shells:

- **sh** o *Bourne **sh**ell*: shell por defecto en las primeras versiones de UNIX
- **bash** o ***B**ourne **a**gain **sh**ell*: versión mejorada de **sh**
  - desarrollada en el proyecto GNU
  - es el shell por defecto en Linux
- **csh** o ***C sh**ell*: desarrollada para UNIX BSD, su sintaxis se basa en la del lenguaje C
- **tcsh** o ***T**urbo **C sh**ell*: versión mejorada de **csh**
- **ksh** o ***K**orn **sh**ell*: basado en Bourne shell con características del C shell

Otros shells:

- **ash** o ***A**lmquist **sh**ell*: clon ligero de **sh** (en Linux Debian, **dash** o *Debian ash*)
- **fish** o **Friendly Interactive Shell**: shell amigable para sistemas UNIX

- **zsh** o ***Z sh****ell*: extensión mejorada de **sh**, incorporando características de otros shells como bash, ksh y tcsh
- **Ion shell**: shell del sistema operativo *Redox*
- **Windows PowerShell**: shell de Microsoft para Windows 7 en adelante

Mas info: Comparison of command shells
Para ver las shells conocidas ver el fichero `/etc/shells`

- El shell por defecto para cada usuario se especifica en el fichero `/etc/passwd`
- Para ver la shell por defecto: `echo $SHELL`
- Para ver la shell actual: `ps | grep $$`
- Para cambiar de shell, ejecutar el comando correspondiente, p.e. `/bin/csh`
  - para volver al shell anterior `exit` o `Ctrl-D`
- Para cambiar la shell por defecto: `chsh`
  - Ejemplo:

```
$ chsh -s /bin/sh pepe
```

En el resto del tema supondremos que usamos **bash**

- usar `bash --version` para ver la versión que tenemos

3.2. La línea de comandos

El shell nos permite enviar comandos al sistema
Los comandos usualmente constan de 3 componentes.

- el nombre del comando (con la ruta absoluta, si no está en el PATH)
- opciones, usualmente precedidas por uno o dos guiones (-)
- argumentos (o parámetros)

Ejemplo: comando `ls` (lista ficheros y directorios)

`$ ls` (lista los archivos del directorio actual)
`$ ls -l` (lista los archivos en formato detallado)
`$ ls -la /tmp` (lista todos los archivos del directorio `/tmp`)

En algunos casos no es necesario usar guion con las opciones, ya que el comando espera por lo menos una:

```
$ tar cf miarchivo.tar arch1 arch2 arch3
```

Comando echo

Imprime el texto o el contenido de las variables:

```
$ echo hola amigo
```

- Comando ⟶ `echo`
- Argumento 1 ⟶ `hola`
- Argumento 2 ⟶ `amigo`

Varios espacios en blanco se interpretan como uno solo

```
$ echo hola            amigo
```

Para que interprete todos los espacios usar comillas simples o dobles

```
$ echo 'hola            amigo'
```

- Comando ⟶ `echo`
- Argumento 1 ⟶ `hola            amigo`

Ejemplos:

```
echo "Hola $USER"
echo -n "Hoy es " ; date
echo -n "Número ficheros en $PWD: ";ls | wc -l
echo -e "uno\tdos\ttres\n"
```

Opciones

- `-n`: elimina el retorno de carro del final
- `-e`: interpreta los caracteres que empiezan por \, como el tabulado (`\t`) o la nueva línea (`\n`)

Caracteres especiales

Hay una serie de caracteres que el shell reconoce y trata de forma especial:

Carácter	Significado
' " \	cambian la forma en que el shell interpreta los caracteres especiales
&	usado después de un comando, indica que se ejecute en background
< > >> << ` |	caracteres de redirección
* ? [ ] [!]	caracteres de substitución (comodines)
$	indica una variable del shell
;	usado para separar múltiples comandos en la misma linea

Ejecución de comandos

Si no se pone la ruta absoluta al comando, este se busca en los directorios indicados en la variable PATH

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
```

- la busqueda en el PATH se realiza secuencialmente en cada directorio listado
- se ejecuta la primera ocurrencia encontrada
- respetar el FHS para ubicar archivos ejecutables
- ¡No poner el `.` en el PATH!

Pueden introducirse múltiples comandos en la misma línea, separados por `;`

```
$ ls -F /etc ; ls -F /usr
```

Un comandos largo puede dividirse en varias líneas mediante `\`

```
$ ls -F /etc \
> /usr
```

Ejecución en segundo plano

Por defecto, los comandos corren en primer plano (*foreground*): el shell espera a que termine el comando antes de aceptar uno nuevo

- para ejecutar un comando en segundo plano (*background*) hacerlo con &

  ```
  $ sleep 10
  $ sleep 10 &
  ```

- para terminar un proceso en foreground `Ctrl-C`
- para pausar un comando en foreground usar `Ctrl-Z`
  - `bg` pasa el proceso a background
  - `fg` lo devuelve a foreground

```
$ sleep 20
Ctrl-Z
[3]+ Stopped     sleep 20
$ bg
[3]+ sleep 20 &
$ fg
sleep 20
```

- el comando `jobs` permite ver la lista de comandos (*jobs*) en background lanzados desde el shell, así como su estado
  - `fg` y `bg` pueden referirse a uno de los jobs

```
$ evince Tema_5.pdf &
$ sleep 100 &
$ jobs
[2]- Running     xpdf Tema_3.pdf &
[3]+ Running     sleep 100 &
$ fg 3
sleep 100
Ctrl-Z
[3]+ Stopped     sleep 100
$ jobs
[2]- Running     xpdf Tema_3.pdf &
[3]+ Stopped     sleep 100
$ bg 3
[3]+ sleep 100 &
$ jobs
[2]- Running     xpdf Tema_3.pdf &
[3]+ Running     sleep 100 &
```

Compleción y edición de comandos

Una de las características de **bash** es la compleción de comandos:

- cuando estamos escribiendo un comando podemos hacer que el sistema complete el nombre pulsando `Tab`
  - si hay varias coincidencias se oye un *beep*, haciendo `Tab` otra vez se ven las posibilidades

bash permite editar los comandos escritos antes de ejecutarlos

- el editor usado para editar los comandos es la *Readline Library*
- por defecto, trabaja en modo `emacs`
  - puede cambiarse la configuración global modificando el fichero `/etc/inputrc` o el fichero `.inputrc` de cada usuario

Algunos comandos de *Readline*

Teclas	**Función**
`Ctrl-b` o ←	Atrás un carácter
`Ctrl-f` o →	Adelante un carácter
`<Backspace>`	Borra el carácter a la izquierda del cursor
`Ctrl-d` o `<SUPR>`	Borra el carácter bajo el cursor
`Ctrl-a` o `<Inicio>`	Va al principio de la línea
`Ctrl-e` o `<Fin>`	Va al final de la línea
`Meta-f`/`Meta-b`	Mueve hacia adelante/atrás una palabra
`Ctrl-k`	Borra desde el cursor al final de la línea
`Meta-d`/`Meta-<Backspace>`	Borra hasta la siguiente/anterior palabra
`Ctrl-l`	Limpia la pantalla

Para más información ver: `info rluserman`

3.3. Histórico de comandos

El **bash** mantiene una lista de comandos previamente usados

- El tamaño de la lista depende de la variable `HISTSIZE` (usar `echo $HISTSIZE` para ver su valor, por defecto es 1000)
- El histórico se almacena en el fichero indicado en la variable `HISTFILE` (por defecto, el fichero `~/.bash_history`[2])

Comandos del historial

[2]El símbolo `~` representa el directorio `HOME` del usuario

Comando	Descripción
`history`	Muestra el historial
`<up-arrow>/<down-arrow>`	Comando anterior/posterior
`!!`	Último comando ejecutado
`!`*n*	*n*-ésimo comando del historial
`!-`*n*	*n* comandos hacia atrás
`!`*cadena*	Último comando ejecutado que empieza por cadena
`!?`*cadena*	Último comando ejecutado que contiene cadena
`^`*cadena1*`^`*cadena2*	Ejecuta el último comando cambiando *cadena1* por *cadena2*
`Ctrl-r`	Busca hacia atrás en el historial
`fc`	Permite ver, editar y reejecutar comandos del historial

Comando interno `fc`

Permite editar comandos del historial y reejecutarlos
Formato:

```
fc [-e ename] [-lnr] [first] [last]
fc -s [pat=rep] [cmd]
```

- En la primera forma, permite editar (con el editor indicado en `ename`, por defecto `nano` o `vi`) los comandos del historial entre el `first` y el `last`
  - `first` y `last` pueden indicar números o caracteres (por los que empiezan los comandos)
  - también admiten números negativos
- Después de editarlos los comandos se reejecutan
- Con la opción `-l` los comandos se listan

```
$ fc -l -12 -10
490      fc
491      echo $HISTFILE
492      less .bash_history
```

- Otras opciones:
  - `-n` suprime los números de comando del listado
  - `-r` muestra los comandos en orden inverso
- Con `-s` el comando `cmd` se reejecuta después de reemplazar cada instancia de `pat` por `rep`

```
$ fc -s FILE=SIZE 2033
echo $HISTSIZE
1000
```

Para más información buscar la parte de `fc` en el manual del **bash**

3.4. Sustitución de nombres de ficheros

Los *comodines* (*wildcards*) permiten especificar múltiples ficheros al mismo tiempo:

> `$ ls -l *html` # Lista los ficheros del directorio actual con terminación html

- también se conoce como *expansión de la shell* o *globbing*
- podemos ver como se hace la expansión poniendo `set -x` o `set -o xtrace`
  - `set +x` para no ver detalles
- podemos desactivar la expansión con `set -f` o `set -o noglob`

Lista de comodines

Carácter	Corresponde a
`*`	0 o más caracteres
`?`	1 carácter
`[ ]`	uno de los caracteres entre corchetes
`[! ]` o `[^ ]`	cualquier carácter que no esté entre corchetes

Ejemplos:

- `cat *`
  muestra el contenido de todos los ficheros en el directorio actual
- `ls a*bc`
  lista todos los ficheros que empiecen por `a` y acaben por `bc` con cero o más caracteres en medio
- `ls a?bc`
  lista todos los ficheros que empiecen por `a` y acaben por `bc` con un carácter cualquiera en medio

- `ls [ic]???`
  lista todos los ficheros que empiecen por `i` o `c` seguido de 3 caracteres cualquiera

- `ls [!ic]???`
  lista todos los ficheros que empiecen por cualquier carácter menos `i` o `c` seguido de 3 caracteres cualquiera

- `ls /etc/[0-9]*`
  lista los ficheros del directorio `/etc` que empiecen por un número

- `ls /tmp/[A-Za-z]*`
  lista los ficheros del directorio `/tmp` que empiecen por una letra mayúscula o minúscula

Si la expansión falla, bash deja el argumento como está

```
$ echo /usr/bin/asdf*fkj
/usr/bin/asdf*fkj
$ ls /usr/bin/asdf*fkj
ls:  /usr/bin/asdf*fkj:  Non hai tal ficheiro ou directorio
```

Los ficheros "ocultos" (que empiezan por `.`) no se expanden

- debemos poner el `.` de forma explícita

Para referirnos a mayúsculas o minúsculas podemos también usar los siguientes patrones:

- `[[:lower:]]`: corresponde a un carácter en minúsculas
- `[[:upper:]]`: corresponde a un carácter en minúsculas
- `[[:alpha:]]`: corresponde a un carácter alfabético
- `[[:digit:]]`: corresponde a un número

Para más detalles: `man 7 glob`

Eliminación del significado especial

bash permite eliminar el significado de los caracteres especiales, usando `'`, `"` o `\`

Carácter	Acción
'	el shell ignora todos los caracteres especiales contenidos entre un par de comillas simples
"	el shell ignora todos los caracteres especiales entre comillas dobles excepto **$**, ` y \
\	el shell ignora el carácter especial que sigue a \

Ejemplos:

```
echo I\'m Pepe
echo "/usr/bin/a*"
echo '$PATH'
echo "$PATH"
```

3.5. Redirección de la entrada/salida

Es posible cambiar la fuente de la entrada o el destino de la salida de los comandos

- toda la E/S se hace a través de ficheros
- cada proceso tiene asociados 3 ficheros para la E/S

Nombre	Descriptor de fichero	Destino por defecto
entrada estándar (*stdin*)	0	teclado
salida estándar (*stdout*)	1	pantalla
error estándar (*stderr*)	2	pantalla

  - por defecto, un proceso toma su entrada de la entrada estándar, envía su salida a la salida estándar y los mensajes de error a la salida de error estándar

  Ejemplo

  ```
  $ ls /bin/bash /kaka
  ls: /kaka: Non hai tal ficheiro ou directorio # Error
  /bin/bash           # Salida estándar
  ```

Para cambiar la entrada/salida se usan los siguientes caracteres:

Carácter	Resultado
`comando < fichero`	Toma la entrada de `fichero`
`comando > fichero`	Envía la salida de `comando` a `fichero`; sobreescribe cualquier cosa de `fichero`
`comando 2> fichero`	Envía la salida de error de `comando` a `fichero` (el 2 puede ser reemplazado por otro descriptor de fichero)
`comando ≫ fichero`	Añade la salida de `comando` al final de `fichero`
`comando ≪ etiqueta`	Toma la entrada para `comando` de las siguientes lineas, hasta una línea que tiene sólo `etiqueta`
`comando 2>&1`	Envía la salida de error a la salida estándar (el 1 y el 2 pueden ser reemplazado por otro descriptor de fichero, p.e. `1>&2`)
`comando &> fichero`	Envía la salida estándar y de error a `fichero`; equivale a `comando > fichero 2>&1`
`comando1 | comando2`	pasa la salida de `comando1` a la entrada de `comando2` (*pipe*)

Ejemplos

- `ls -l > lista.ficheros`
  Crea el fichero `lista.ficheros` conteniendo la salida de `ls -l`
- `ls -l /etc ≫ lista.ficheros`
  Añade a `lista.ficheros` el contenido del directorio `/etc`
- `cat < lista.ficheros | more`
  Muestra el contenido de `lista.ficheros` página a página (equivale a `more lista.ficheros`)
- `ls /kaka 2> /dev/null`
  Envía los mensajes de error al dispositivo nulo (a la *basura*)
- `> kk`
  Crea el fichero `kk` vacío
- `cat > entrada`
  Lee información del teclado, hasta que se teclea `Ctrl-D`; copia todo al fichero entrada
- `cat ≪ END > entrada`
  Lee información del teclado, hasta que se introduce una línea con `END`; copia todo al fichero `entrada`
- `ls -l /bin/bash /kaka > salida 2> error`
  Redirige la salida estándar al fichero `salida` y la salida de error al fichero `error`

- `ls -l /bin/bash /kaka > salida.y.error 2>&1`
  Redirige la salida estándar y de error al fichero `salida.y.error`; el orden es importante:

  ```
  ls -l /bin/bash /kaka 2>&1 > salida.y.error
  ```

  no funciona, por qué?

- `ls -l /bin/bash /kaka &> salida.y.error`
  Igual que el anterior

- `cat /etc/passwd > /dev/pts/2`
  Muestra el contenido de `/etc/passwd` en el terminal `pts/2`

  - usar el comando `tty` para ver el nombre del terminal en el que estamos

Comandos útiles con pipes y redirecciones

1. `tee`

   - copia la entrada estándar a la salida estándar y también al fichero indicado como argumento:
     - `ls -l | tee lista.ficheros | less`
       Muestra la salida de `ls -l` página a página y la almacena en `lista.ficheros`
   - Opciones:
     - `-a`: no sobreescribe el fichero, añade al final

2. `xargs`

   - permite pasar un elevado número de argumentos a otros comandos
   - lee la entrada estándar, y ejecuta el comando uno o más veces, tomando como argumentos la entrada estándar (ignorando líneas en blanco)
   - Ejemplos:

     ```
     $ locate README | xargs cat | fmt -60 >\
     /home/pepe/readmes
     ```
     `locate` encuentra los ficheros `README`; mediante `xargs` los ficheros se envían a `cat` que muestra su contenido; este se formatea a 60 caracteres por fila con `fmt` y se envía al fichero

`readmes`

`$ locate README | xargs -I {} cp {} /tmp/`
copia los `README` en el directorio `/tmp`; la opción `-I {}` permite que {} sea reemplazado por los nombres de los ficheros

3.6. Variables de shell

Uso de variables:

- control del entorno (*environment control*)
- programación shell

Dos tipos

- variables locales: visibles sólo desde el shell actual
- variables globales o de entorno: visibles en todos los shells

El comando `set` permite ver las variables definidas en nuestra shell

- El nombre de las variables debe:
  - empezar por una letra o _
  - seguida por cero o mas letras, números o _ (sin espacios en blanco)

Uso de las variables

- Asignar un valor: *nombre_variable=valor*

```
$ una_variable=hola
$ un_numero=15
$ nombre="Pepe Pota"
```

  Los espacios en blanco son tenidos en cuenta:
  - usar comillas para incluirlos en la variable

- Acceder a las variables: *${nombre_variable}* o *$nombre_variable*

```
$ nombre="Pepe Pota"
$ echo $nombre
Pepe Pota
$ echo ${nombre}mo
```

```
Pepe Potamo
$ comando=ls
$ $comando
20041020    DeadLetters    ioports.txt    news
```

- Vaciar una variable: *nombre_variable=*

```
$ nombre=
$ echo ${nombre}mo
mo
```

- Eliminar una variable: *unset nombre_variable*

```
$ unset nombre
$ echo ${nombre}mo
mo
```

- variables de solo lectura: *readonly nombre_variable*

```
$ readonly nombre
$ unset nombre
bash: unset: nombre: cannot unset: readonly variable
```

  - `readonly` sin parámetros, muestra las variables de solo lectura definidas

Variables de entorno

Cada shell se ejecuta en un *entorno* (*environment*)

- el entorno de ejecución especifica aspectos del funcionamiento del shell
- esto se consigue a través de la definición de variables de entorno (o variables globales)
- algunas variables son:

Nombre	Propósito
HOME	directorio base del usuario
SHELL	el ejecutable para la shell que estamos usando
USERNAME	el nombre de usuario
PWD	el directorio actual
PATH	el *path* para los ejecutables
MANPATH	el *path* para las páginas de manual
PS1/PS2	*prompts* primario y secundario
LANG	aspectos de localización geográfica e idioma
LC_*	aspectos particulares de loc. geográfica e idioma

- para ver las variables de entorno usar `env` o `printenv`
  - `env` permite ejecutar un programa en un entorno modificado, p.e.

    ```
    $ env -i bash
    ```

    inicia una shell en un entorno limpio

Definición de variables de entorno

Para definir una nueva variable de entorno: `export`

- Ejemplo:

```
$ nombre="Pepe Pota"        # Define una variable de shell
$ echo $nombre              # Úsala
Pepe Pota
$ bash                      # Inicia un nuevo shell
$ echo Mi nombre es $nombre # Intenta usar la variable
Mi nombre es                # del shell padre
Mi nombre es
$ exit                      # Vuelve al shell padre
$ echo $nombre              # Usa la variable en el shell
Pepe Pota                   # padre
$ export nombre             # Exporta la variable
$ bash                      # Inicia un nuevo shell
$ echo Mi nombre es $nombre # Intenta usar la variable
Mi nombre es Pepe Pota      # del shell padre
```

- La variable exportada (variable de entorno) es visible en el shell hijo
  - el shell hijo crea una copia local de la variable y la usa
  - las modificaciones de esa copia no afectan al shell padre

```
$ export nombre="Pepe Pota"
$ bash
$ nombre="Tico Mico"        # El hijo cambia la variable
$ exit
$ echo Mi nombre es $nombre # No hay cambio en el padre
Mi nombre es Pepe Pota
```

El prompt

El aspecto del prompt se define a través de las variables `PS1` y `PS2`

- `PS1`: prompt principal
- `PS2`: prompt secundario
  - aparece cuando un comando se escribe en varias líneas
- Ejemplo:

```
$ PS1="[\u@\h:\w]\$ "
[tomas@jumilla:/usr/bin]$ PS2="(cont.)\\"
[tomas@jumilla:/usr/bin]$ ls \
(cont.)\ -la
```

- Mas complejo:

```
$ PS1="\[\033[1;32m\](\d \u@\h:\W)\n\[\033[m\](\t)$"
(Xov Xul 21 tomas@jumilla:bin)
(12:34:56)$
```

- Pueden usarse diferentes códigos para configurar el prompt:

Código	Significado
`\u`	nombre del usuario
`\h`	nombre de la máquina
`\w`	nombre completo del directorio actual
`\W`	base del nombre del directorio actual
`\$`	muestra $ para usuarios normales y # para el root
`\!`	numero en la historia del comando actual
`\#`	número del comando actual
`\d`	fecha actual
`\t`	hora actual
`\s`	nombre del shell
`\n`	nueva línea
`\\`	una \
`$(comando)`	salida del comando
`\[`	comienza una secuencia de caracteres no imprimibles
`\]`	termina una secuencia de caracteres no imprimibles
`\`*nnn*	carácter ASCII correspondiente al octal *nnn*

Ejemplo: prompt con color

```
\[\033[BRIGHTORNOT;COLORNUMBERm\]
```

donde:

- `BRIGHTORNOT`: 0 (sin brillo); 1 (brillante)
- `COLORNUMBER`: número 30-39, siendo
  - 30: Negro/gris oscuro
  - 31: Rojo
  - 32: Verde
  - 33: Amarillo
  - 34: Azul
  - 35: Magenta
  - 36: Fucsia
  - 37: Blanco/gris claro
  - 38: Subrayado
  - 39: Por defecto
- Ejemplo: `\[\033[1;32m\]`

Para cancelar el efecto de color: `\[\033[m\]`

Sustitución avanzada de variables

El shell proporciona un conjunto adicional de constructores complejos para la substitución variables:

Constructor	Propósito
`${variable:-valor}`	reemplaza el constructor con el valor de la variable si tiene uno, si no, usa `valor` pero no hace `variable=valor`
`${variable:=valor}`	igual que el anterior, pero si `variable` no tiene valor, asígnale `valor`
`${variable:?mensaje}`	reemplaza el constructor con el valor de la variable si tiene uno, si no, muestra `mensaje` en *stderr* (si se omite `mensaje`, muestra un mensaje de error en *stderr*)
`${variable:+valor}`	si `variable` tiene valor lo reemplaza con `valor`, si no, no hace nada; no asigna `variable=valor`

Ejemplo:

```
$ miNombre=
$ echo Mi nombre es $miNombre
Mi nombre es
$ echo Mi nombre es ${miNombre:-"SIN NOMBRE"}
Mi nombre es SIN NOMBRE
$ echo Mi nombre es $miNombre
Mi nombre es
$ echo Mi nombre es ${miNombre:="SIN NOMBRE"}
Mi nombre es SIN NOMBRE
$ echo Mi nombre es $miNombre
Mi nombre es SIN NOMBRE
```

Ejemplo (cont.):

```
$ suNombre=
$ echo Su nombre es ${suNombre:?"no tiene nombre"}
bash: suNombre: no tiene nombre
$ echo Su nombre es ${suNombre:?}
bash: suNombre: parameter null or not set
$ echo Su nombre es ${suNombre:+"SIN NOMBRE"}
Su nombre es
$ suNombre="Pepe Potamo"
$ echo Su nombre es ${suNombre:+"SIN NOMBRE"}
Su nombre es SIN NOMBRE
$ echo Su nombre es ${suNombre}
Su nombre es Pepe Potamo
```

3.7. Sustitución de comandos

Permite que la salida de un comando reemplace el propio comando
Formato:

`$(comando)` o `` `comando` ``

Ejemplos:

```
$ echo date
date
$ echo `date`
Xov Xul 21 13:09:39 CEST 2005
$ echo líneas en fichero=$(wc -l fichero)
```

`wc -l` cuenta el número de líneas en el fichero; el comando se ejecuta y su salida se pasa al `echo`

3.8. Orden de evaluación

Desde que introducimos un comando hasta que se ejecuta, el shell ejecuta los siguientes pasos, y en el siguiente orden:

1. Redirección E/S
2. Sustitución (expansión) de variables: reemplaza cada variable por su valor
3. Sustitución (expansión) de nombres de ficheros: sustituye los comodines por los nombres de ficheros

Si no se tiene en cuenta ese orden, pueden aparecer problemas:

```
$ star=\*
$ ls -d $star
cuatro  dos  tres  uno
$ pipe=\|
$ cat uno $pipe more
cat: |: Non hai tal ficheiro ou directorio
cat: more: Non hai tal ficheiro ou directorio
```

Comando `eval`

Evalúa la línea de comandos 2 veces:

- la primera hace todas las substituciones

- la segunda ejecuta el comando

Ejemplo:

```
$ pipe=\|
$ eval cat uno $pipe more
Este es el fichero uno
...
```

- En la primera pasada reemplaza `$pipe` por |
- En la segunda ejecuta el comando `cat uno | more`

3.9. Otras expansiones

Además de las ya vistas, bash permite otras expansiones:

1. Expansión de llaves
2. Expansión de la tilde
3. Expansión aritmética

Para más detalles sobre la expansión del shell mirar el manual de bash, sección `EXPANSION`

Expansión de llaves

Permite generar strings arbitrarios

- no tiene para nada en cuenta los ficheros existentes en el directorio actual

```
$ echo a{d,c,b}e
ade ace abe
```

Expansión de la tilde

Expande la tilde como directorio HOME del usuario indicado

- si no se indica usuario, usa el usuario actual

```
cd ~              # Accedemos al nuestro HOME
cd ~root          # Accedemos al HOME de root
ls ~pepe/cosas/   # Vemos el contenido del directorio
                  cosas de pepe
```

Expansión aritmética

Permite evaluar expresiones aritméticas enteras

- se usa `$(( expresión ))` o `$[ expresión ]`
- `expresión` tiene una sintaxis similar a la del lenguaje C
  - permite operadores como `++`, `+=`, `&&`,...
- También se puede usar `let`

```
$ let numero=(numero+1)/2  #usar " si se dejan espacios en blanco
```

- Ejemplos:

```
$ echo $(((4+11)/3))
5
$ numero=15
$ echo $((numero+3))
18
$ echo $numero
15
$ echo $((numero+=4))
19
$ echo $numero
19
$ numero=$(((numero+1)/2))
$ echo $numero
10
```

Otras formas de hacer operaciones aritméticas

- Comando `expr`: permite evaluar expresiones enteras

```
$ expr 5 + 6 \* 10
65
```

  - los parámetros deben separarse por espacios
  - se debe *escapar* el *, para que no lo tome por un comodín
  - ver `man expr` para opciones

- Comando `bc`: expresiones con números de precisión arbitraria

```
$ pi=$(echo "scale=10; 4*a(1)" | bc -l)
$ echo $pi
3.1415926532
```

3.10. Alias

Permiten reemplazar un string por una palabra cuando se usa como la primera palabra de un comando simple

```
$ alias cl='cat /tmp/lista'
$ cl
Esto es una lista
```

Se usa para obtener nombre simples para los comandos más comunes:

```
$ alias ls='ls -F'
$ alias l='ls -la'
$ alias l='rm -i'
```

Si usamos el comando interno `alias` sin opciones nos devuelve la lista de alias definidos
Para eliminar un alias: `unalias`

```
$ unalias l
```

3.11. Ficheros de inicialización de bash

Cuando se inicia bash se leen automáticamente distintos ficheros de inicialización

- En estos ficheros el usuario define variables de entorno, alias, el prompt, el path, etc.
- Los ficheros que se leen dependen de la forma de invocar bash

Formas de invocar bash:

1. Invocado como un *login shell* interactivo
   - cuando entramos en el sistema con login y password, usamos `su -`, o iniciamos bash con la opción `--login`
   - cuando se inicia, se leen los siguientes ficheros:
     (a) `/etc/profile`
     (b) el primero que exista de : `~/.bash_profile`, `~/.bash_login` o `~/.profile`
   - al dejar el shell se lee `~/.bash_logout`
2. Invocado como un *non-login shell* interactivo

- cuando lo iniciamos sin opciones (bash), abrimos una nueva ventana de comandos (entramos sin login ni password), o usamos `su`
- se leen los ficheros:
  (a) `/etc/bash.bashrc`
  (b) `~/.bashrc`[3]
- al salir no se ejecuta nada

3. Invocado como un shell no interactivo
   - por ejemplo, cuando se lanza un script
   - en un shell no interactivo, la variable `$PS1` no está disponible
   - se lee el fichero definido en la variable `BASH_ENV`

3.12. Otras opciones de bash

Algunas opciones del bash pueden cambiarse con `set`

- ya comentados `set -o noglob` y `set -o xtrace`

Las opciones se habilitan con `-o` y se deshabilitan con `+o`

- Ejemplos:
  - `noclobber`: impide que un fichero sea sobreescrito por una operación de redirección

```
$ set -o noclobber     # Equivale a set -C
$ touch test
$ date >test
bash:  test:  cannot overwrite existing file
$ set +o noclobber     # Equivale a set +C
$ date >test
```

  - `nounset`: trata las variables no definidas como error

```
$ echo $VAR

$ set -o nounset     # Equivale a set -u
$ echo $VAR
bash:  VAR: unbound variable
```

[3]Usualmente, desde `.bash_profile` se invoca al `bashrc` de la siguiente forma:
`if [ -f ~/.bashrc ]; then . ~/.bashrc; fi`

Para ver el estado de las diferentes opciones usar `set -o`

- para más información, ver en la página de manual de bash, sección `SHELL BUILTIN COMMANDS`, o en el manual de `bash-builtins`, el apartado sobre `set`

Comando interno `shopt`

Permite controlar comportamientos opcionales del shell

- sin opciones, muestra una lista de características indicando si están activas (`on`) o no (`off`)
- opciones:
  - `-s` *`optname`* activa la opción indicada
  - `-u` *`optname`* desactiva la opción indicada
- Ejemplo:

```
$ touch hola Hola
$ ls h*
hola
$ shopt -s nocaseglob
$ ls h*
hola Hola
```

- para más información, manual de `bash-builtins`

3.13. Prácticas: La línea de comandos

Introducción al *shell*

1. Ver la shell que estamos usando
2. Cambiar temporalmente la shell por la shell **sh**. Comprobar que deja de funcionar la repetición de comandos y el histórico.
3. Volver al shell anterior.

Uso de la línea de comandos

1. Probar la ejecución de los comandos `ls` y `echo`
   - probar a lanzar varios comandos en la misma línea, con `;` y `\`
2. Utilizar el comando `sleep` para probar la ejecución en primer y segundo plano
   - usar `jobs`, `fg` y `bg`
3. Probar los diferentes comandos de *Readline*

Histórico de comandos

1. Probar los diferentes comandos del historial
2. Usar `fc` para editar el historial

Sustitución de nombres de ficheros

1. Crear los siguientes ficheros en un subdirectorio:

   feb86 Feb86 en12.89 en19.89 en26.89 en5.89 en85 en86 en87 en88 En88 mar88 memo1 memo10 memo2 memo2.sd .memo

   - el comando `touch` permite crear ficheros vacíos
2. Probar los siguientes comandos:
   (a) `echo *`
   (b) `echo .*`
   (c) `echo *[!0-9]`
   (d) `echo m[a-df-z]*`
   (e) `echo en*`
   (f) `echo *.*`
   (g) `echo ?????`
   (h) `echo *89`
   (i) `echo en?? feb?? mar??`
   (j) `echo [fjm][ae][bnr][0-9][0-9]`
   (k) `echo E*`

(l) `echo [[:upper:]]*`

(m) `echo [E-F]*`

3. Crear ficheros con los siguientes nombres, y después borrarlos

```
memo*
hola amigo migo
"adios"
-top
```

   - en este último caso, el - confunde al shell, ya que piensa que es una opción del comando: probar usando -- después del comando, para indicar que ya no hay más opciones

Redirección de la entrada/salida

1. Probar los ejemplos de los apumtes.
2. Escribe un comando que permita determinar el número de ficheros de un directorio
   - usa el comando `wc -l` que permite obtener el número de líneas de un fichero

Variables de shell

1. Probar los ejemplos de los apuntes
2. Prueba a cambiar alguna variable de entorno
   - cambia la variable `LC_TIME` a C y mira como muestra la fecha (comando `date`)
3. Definir un prompt con dos lineas:
   - en la primera, que aparezca la fecha, la hora y el shell, en color rojo
   - y en la segunda, en nombre de usuario, de la máquina y el directorio actual
   - Por ejemplo:

```
Día Mér Xul 20 Hora 20:13:10 - Shell bash
[tomas@pc1:/usr/local/bin]$
```

Sustitución de comandos

1. Ejecuta `date` y guarda el resultado en un fichero cuyo nombre se construye en base a nuestro nombre de usuario y al año de la fecha actual, p.e. fichero-tomas-2005
   - para obtener el nombre de usuario usa el comando `whoami`
   - para obtener el año: `date +%Y`

Orden de evaluación y otras expansiones

1. Probar los ejemplos de los apuntes con `eval` y otras expansiones
2. Leer la sección `EXPANSION` de manual del bash

Alias

1. Ver los alias que tenemos definidos por defecto
2. Hacer un comando (con un alias) que nos indique el número de ficheros en el directorio `/tmp`
3. Modifica el fichero de inicio para cambiar tu prompt y añadir algunos alias

4. Comandos para el procesamiento de textos

Existen una serie de comandos para manejar ficheros de texto, como `tac`, `rev`, `nl`, `head`, `tail`, `sort`, `uniq`, `expand`, `fmt`, `cut`, `paste`, `tr`, `join`, `split`, `wc`, `od`, `grep`, `sed` o `awk`

- también se conocen como *filtros*: obtienen su entrada de la entrada estándar (o un fichero) y envían la salida a la salida estándar:

```
sort < archivo.txt | head -3 > otro_archivo.txt
```

- casi todos estos comandos tienen, entre otras opciones, las siguientes dos:
  - `--help` muestra una pequeña ayuda y sal
  - `--version` muestra la versión del comando y sal
- también podemos saber más del comando a través de la página de manual o de `info`

4.1. Resumen de comandos

- Buscar patrones en un texto: `grep`
- Buscar y reemplazar: `sed`
- Ordenar las líneas alfabéticamente: `sort`
- Escribir partes seleccionadas de un fichero a la salida estándar: `cut`
- Unir texto de varios ficheros: `paste`
- Formatear párrafos: `fmt`
- Borrar y/o reemplazar caracteres: `tr`
- Eliminar líneas repetidas: `uniq`
- Combinar varios ficheros: `join`
- Dividir un fichero en ficheros más pequeños: `split`
- Mostrar el principio/final de un fichero: `head`/`tail`
- Mostrar el fichero al revés: `tac`, `rev`
- Mostrar el número de líneas, palabras y bytes de un fichero: `wc`
- Añadir números de línea: `nl`
- Convertir `TABs` en espacios: `expand`
- Mostrar un fichero en diferentes formatos: `od`

Comentaremos brevemente cada uno de ellos

grep

muestra las líneas de un fichero que concuerdan con un patrón

Ejemplo:

```
$ cat nombres.txt
María Martín
Luis Andión
Martín Gómez
Jorge Pena
$ grep 'Martín' nombres.txt
María Martín
Martín Gómez
```

El patrón se puede expresar como una expresión regular

sed

busca y sustituye un patrón en un fichero

Ejemplo:

```
$ cat nombres.txt
María Martín
Luis Andión
Martín Gómez
Jorge Pena
$ sed 's/Martín/Tirman/' nombres.txt
María Tirman
Luis Andión
Tirman Gómez
Jorge Pena
```

El patrón se puede expresar como una expresión regular

sort

ordena alfabéticamente líneas de texto y las muestra en la salida estándar

Formato:

`sort` [opciones] *fichero*

Algunas opciones:

- `-b` ignora blancos al principio de línea
- `-f` no distingue mayúsculas/minúsculas
- `-r` orden inverso
- `-m` mezcla ficheros previamente ordenados
- `-n` ordena numéricamente
- `-k` *POS1[, POS2]* ordena según los campos desde *POS1* o *POS2*, o el final si no está *POS2* (el primer campo es 1)

Ejemplos:

```
$ cat nombres.txt
María Pérez
luis Andión
Adriana Gómez
jorge pena
$ sort nombres.txt
Adriana Gómez
María Pérez
jorge pena
luis Andión

$ sort -f nombres.txt
Adriana Gómez
jorge pena
luis Andión
María Pérez
$ sort -f +1 +0 nombres.txt #Obsoleto (no usar)
luis Andión
Adriana Gómez
jorge pena
María Pérez
$ sort -f -k 2,2 nombres.txt
luis Andión
Adriana Gómez
jorge pena
María Pérez
```

cut

Escribe partes seleccionadas de un fichero a la salida estándar; puede usarse para seleccionar columnas o campos de un fichero específico

Formato:

`cut` [opciones] *fichero*

Algunas opciones:

- `-b`, `-c`, `-f` corta por bytes, caracteres o campos, respectivamente
- `-d` fija el carácter delimitador entre campos (por defecto, `TAB`)

Ejemplos:

```
$ cat nombres-ord.txt
Luis Andión
Adriana Gómez
Jorge Pena
María Pérez

$ cut -c 1-7 nombres-ord.txt
Luis An
Adriana
Jorge P
María P
$ cut -c 1-5,9-10 nombres-ord.txt
Luis ió
AdriaGó
Jorgena
Maríare
$ cut -d ' ' -f 1 nombres-ord.txt
Luis
Adriana
Jorge
María
```

paste

Permite unir texto de varios ficheros, uniendo las líneas de cada uno de los ficheros

Formato:

`paste` [opciones] *fichero1 [fichero2] ...*

Algunas opciones:

- `-s` pega los ficheros secuencialmente, en vez de intercalarlos
- `-d` especifica los carácteres delimitadores en la salida (por defecto, `TAB`)

Ejemplos:

```
$ cat nombres.txt
Luis
Adriana
Jorge
María
$ cat apellidos.txt
Andión
Gómez
Pena
Pérez
$ paste nombres.txt apellidos.txt
Luis      Andión
Adriana   Gómez
Jorge     Pena
María     Pérez
$ paste -d ' ' nombres.txt apellidos.txt
Luis Andión
Adriana Gómez
Jorge Pena
María Pérez
$ paste -s -d '\t\n' nombres.txt
Luis    Adriana
Jorge   María
```

fmt

Formatea cada párrafo, uniendo o separando líneas para que todas tengan el mismo tamaño

Algunas opciones:

- `-`*n* o `-w` *n* pone la anchura de las líneas a *n* (por defecto, 75)
- `-c` conserva la indentación a principio de línea y alinea a la izquierda la segunda línea
- `-s` las líneas pueden dividirse, no unirse
- `-u` uniformiza el espaciado entre palabras

Ejemplo:

```
$ cat quijote.txt
En un lugar de la Mancha, de    cuyo nombre no
quiero acordarme, no ha mucho tiempo
que vivía un
hidalgo    de los de lanza en astillero, adarga
antigua, rocín flaco y galgo corredor.

$ fmt -w 45 -u quijote.txt
En un lugar de la Mancha, de cuyo nombre
no quiero acordarme, no ha mucho tiempo
que vivía un hidalgo de los de lanza en
astillero, adarga antigua, rocín flaco y
galgo corredor.
```

tr

Borra caracteres o reemplaza unos por otros

Formato:

`tr` [opciones] *set1 set2*

Algunas opciones:

- `-d` borra los caracteres especificados en *set1*
- `-s` reemplaza caracteres repetidos por un único carácter

Ejemplos:

```
$ tr 'a-z' 'A-Z' < quijote.txt
EN UN LUGAR DE LA MANCHA, DE CUYO NOMBRE...
$ tr -d ' ' < quijote.txt
EnunlugardelaMancha,decuyonombre...
$ tr au pk < quijote.txt
En kn lkgpr de lp Mpnchp, de ckyo nombre...
$ tr lcu o < quijote.txt | tr -s o
En on ogar de oa Manoha, de oyo nombre
```

uniq

Descarta todas (menos una) las líneas idénticas sucesivas en el fichero

Formato:

`uniq` [opciones] *fichero*

Algunas opciones:

- `-d` muestra las líneas duplicadas (sin borrar)
- `-u` muestra sólo las líneas sin duplicación
- `-i` ignora mayúsculas/minúsculas al comparar
- `-c` muestra el número de ocurrencias de cada línea
- `-s` n no compara los n primeros caracteres
- `-f` n no compara los n primeros campos
- `-t` c usa el carácter c como separador de campos (por defecto, espacio o tabulado)

Ejemplo:

```
$ cat nombres.txt
Julio Lorenzo
Pedro Andión
Celia Fernández
Celia Fernández
Juan Fernández
```

```
Enrique Pena
$ uniq nombres.txt
Julio Lorenzo
Pedro Andión
Celia Fernández
Juan Fernández
Enrique Pena
$ uniq -f 1 -c nombres.txt
  1 Julio Lorenzo
  1 Pedro Andión
  3 Celia Fernández
  1 Enrique Pena
```

join

Permite combinar dos ficheros usando campos: busca en los ficheros por entradas comunes en el campo y une las líneas; los ficheros deben estar ordenados por el campo de unión

Formato:

`join` [opciones] *fichero1 fichero2*

Algunas opciones:

- `-i` ignora mayúsculas/minúsculas
- `-1` *FIELD* une en el campo *FIELD* (entero positivo) de *fichero1*
- `-2` *FIELD* une en el campo *FIELD* de *fichero2*
- `-j` *FIELD* equivalente a `-1` *FIELD* `-2` *FIELD*
- `-t` *CHAR* usa el carácter *CHAR* como separador de campos
- `-o` *FMT* formatea la salida (*M.N* fichero *M* campo *N*, 0 campo de unión)
- `-v` *N* en vez de la salida normal, muestra las líneas que no se unen del fichero *N*
- `-a` *N* además la salida normal, muestra las líneas que no se unen del fichero *N*

Ejemplo:

```
$ cat nombres1.txt
Luis Andión
Adriana Gómez
Jorge Pena
María Pérez
$ cat nombres2.txt
Pedro Andión
Celia Fernández
Julio Lorenzo
Enrique Pena
$ join -j 2 nombres1.txt nombres2.txt
Andión Luis Pedro
Pena Jorge Enrique
$ join -j 2 -o 1.1 2.1 0 nombres1.txt nombres2.txt
Luis Pedro Andión
Jorge Enrique Pena
```

split

Divide un fichero en ficheros más pequeños; los ficheros más pequeños se nombran a partir del *prefijo* especificado (*prefijo*aa, *prefijo*ab,...)

Formato:

`split` [opciones] *fichero prefijo*

Si no se pone *fichero*, o se pone `-` se lee la entrada estándar

Algunas opciones:

- `-l` n pone n lineas en cada fichero de salida (por defecto 1000)
- `-b` n pone n bytes en cada fichero de salida
- `-C` n pone en cada fichero de salida tantas líneas completas como sea posible sin sobrepasar n bytes
- `-d` usa números en vez de letras para el nombre de los ficheros de salida

Ejemplo:

```
$ split -l 2 quijote.txt quij
$ ls quij*
quijaa quijab quijac quijote.txt
$ cat quijaa
En un lugar de la Mancha, de cuyo nombre
no quiero acordarme, no ha mucho tiempo
$ cat quijac
galgo corredor.
$ split -l 2 -d quijote.txt quij
$ ls quij*
quij00 quij01 quij02 ...
```

head

Muestra el principio de un fichero

Formato:

`head` [opciones] *fichero*

Algunas opciones:

- `-n` N ó `-`N muestra las primeras N líneas
- `-c` N muestra los primeros n bytes
- `-v` le añade una línea de cabecera, con el nombre del fichero

Ejemplo:

```
$ head -n 2 -v quijote.txt
==>quijote.txt <==
En un lugar de la Mancha, de cuyo nombre
no quiero acordarme, no ha mucho tiempo
```

tail

Muestra el final de un fichero

Algunas opciones:

- `-n` N ó `-`N muestra las últimas N líneas (por defecto, 10)
- `+`N muestra de la línea N al final
- `-c` N muestra los últimos N bytes
- `-f` hace que `tail` corra en un lazo, añadiendo líneas a medida que el fichero crece (útil para cuando queremos ver como se modifica un fichero)
- `--retry` útil con `-f`; aunque el fichero no exista o sea inaccesible continua intentando hasta que puede abrirlo
- `-v` le añade una línea de cabecera, con el nombre del fichero

Ejemplo:

```
$ tail -n 2 -v quijote.txt
==>quijote.txt <==
astillero, adarga antigua, rocín flaco y
galgo corredor.
```

tac, rev

`tac` imprime el fichero de la última a la primera línea (opuesto a `cat`); `rev` invierte las lineas del fichero

Ejemplos:

```
$ tac quijote.txt
galgo corredor.
astillero, adarga antigua, rocín flaco y
que vivía un hidalgo de los de lanza en
no quiero acordarme, no ha mucho tiempo
En un lugar de la Mancha, de cuyo nombre

$ rev quijote.txt
erbmon oyuc ed ,ahcnaM al ed ragul nu nE
opmeit ohcum ah on ,emradroca oreiuq on
ne aznal ed sol ed ogladih nu aíviv euq
y ocalf nícor ,augitna agrada ,orellitsa
.roderroc oglag
```

wc

Muestra el número de líneas, palabras y bytes de un fichero

Formato:

`wc` [opciones] *fichero*

Algunas opciones:

- `-l` muestra sólo el número de lineas
- `-w` muestra sólo el número de palabras
- `-c` muestra sólo el número de bytes
- `-L` muestra la longitud de la línea más larga

Ejemplo:

```
$ wc quijote.txt
  5 33 178 quijote.txt
$ wc -l quijote.txt
5 quijote.txt
$ wc -w quijote.txt
33 quijote.txt
$ wc -c quijote.txt
178 quijote.txt
```

nl

Añade números de línea; `nl` considera los ficheros separados en *páginas lógicas*, cada una de ellas con una cabecera, cuerpo y pie, cada una de estas secciones se numera de forma independiente, y la numeración se reinicia para cada página; los comienzos de cabecera, cuerpo y pie de cada página se marcan, respectivamente, con `\:\:\:`, `\:\:` y `\:`

Formato:

`nl` [opciones] *fichero*

Algunas opciones:

- `-b`, `-h` o `-f` *ESTILO* indica el estilo de numeración para cuerpo, cabecera o pie, que puede ser:
  - `a`: numera todas las líneas
  - `t`: numerar sólo las líneas no vacías (por defecto para el cuerpo)
  - `p` *REGEXP*: numera sólo las líneas que concuerdan con *REGEXP*
  - `n`: no numera ninguna línea (por defecto para cabecera y pie)
- `-v` *n* inicia la numeración en *n* (por defecto, 1)
- `-i` *n* incrementa los números por *n* (por defecto, 1)
- `-p` no reinicia la numeración al principio de cada página
- `-s` *STRING* una STRING para separar los números de línea del texto (por defecto ' ')

Ejemplo:

```
$ nl -s 'q ' quijote.txt
  1q En un lugar de la Mancha, de cuyo nombre
  2q no quiero acordarme, no ha mucho tiempo
  3q que vivía un hidalgo de los de lanza en
  4q astillero, adarga antigua, rocín flaco y
  5q galgo corredor.
```

expand

Convierte `TABs` en espacios; útil debido a que la representación del `TAB` puede ser diferente en distintos sistemas

Formato:

`expand` [opciones] *fichero* ...

Algunas opciones:

- `-t` *n* reemplaza cada `TAB` por *n* espacios (por defecto, 8)
- `-i` solo reemplaza los `TABs` de principio de línea

Ejemplos:

```
$ cat hola.c
main() {
        for(i=0; i<10;i++)
                printf("Hola mundo!\n");
}
$ expand -t 2 hola.c
main() {
  for(i=0; i<10;i++)
    printf("Hola mundo!\n");
}
```

El comando `unexpand` hace la operación contraria

od

Muestra un fichero en octal, hexadecimal o otros formatos; en cada línea muestra (en la primera columna) el *offset*

Formato:

`od` [opciones] *fichero*

Algunas opciones:

- `-t` *TIPO* especifica el formato de la salida (por defecto octal): `o` para octal, `x` para hexadecimal, `d` para decimal, `c` para caracteres ASCII, `a` para caracteres con *nombre*...
- `-A` *TIPO* especifica el formato del offset (por defecto octal): `o`, `x`, `d` como antes, `n` para que no aparezca
- `-w` *BYTES* número de bytes por linea (por defecto 16)

Ejemplo:

```
$ od -t x -A x quijote.txt
000000 75206e45 756c206e 20726167 6c206564
000010 614d2061 6168636e 6564202c 79756320
000020 6f6e206f 6572626d 206f6e0a 65697571
...
```

5. Programación Shell-Script

Bash (y otros *shells*) permiten programar *scripts*:

Script o programa *shell*: fichero de texto conteniendo comandos externos e internos, que se ejecutan línea por línea

- El programa puede contener, además de comandos
  1. variables
  2. constructores lógicos (`if...then`, `AND`, `OR`, etc.) y lazos (`while`, `for`, etc.)
  3. funciones
  4. comentarios

Para saber más:

- *Advanced Bash-Scripting Guide*, Mendel Cooper, Última revisión Marzo 2014, www.tldp.org/guides.html
- Artículos sobre *Introduction to Shell Scripting*, Ben Okopnik, en Linux Gazette, linuxgazette.net/issue52/okopnik2.html linuxgazette.net/issue53/okopnik.html y linuxgazette.net/issue54/okopnik.html
- *The Deep, Dark Secrets of Bash*, Ben Okopnik, Linux Gazette, linuxgazette.net/issue55/okopnik.html

5.1. Ejecución de un script

Los scripts deben empezar por el *Shebang* `#!` seguido del programa a usar para interpretar el script:

- `#!/bin/bash` script de bash
  - Esto supone que el comando `bash` está en `/bin`
  - Mejor usar: `#!/usr/bin/env bash`
- `#!/bin/sh` script de shell
- `#!/usr/bin/env python` script de Python

Las forma usuales de ejecutar un script es:

- darle permiso de ejecución al fichero y ejecutarlo como un comando:

```
$ chmod +x helloworld
./helloworld
```

- ejecutar una shell poniendo como argumento el nombre del script (sólo necesita permiso de lectura)

```
$ bash helloworld
```

Ejecución con la shell actual

Los métodos anteriores arrancan una sub-shell que lee las ordenes del fichero, la ejecuta y después termina cediendo el control nuevamente a la shell original Existe una forma de decirle a la shell actual que lea y ejecute una serie de ordenes por si misma sin arrancar una sub-shell:

```
$ . helloworld
```

o bien:

```
$ source helloworld
```

Ejemplo:

```
$ cat shellid.sh
#!/bin/bash
echo "Shell ejecutando el script, PID = $$"
$ echo "PID actual = $$"
PID actual = 6919
$ bash shellid.sh
Shell ejecutando el script, PID = 26824
$ . shellid.sh
Shell ejecutando el script, PID = 6919
```

5.2. Paso de parámetros

Es posible pasar parámetros a un scripts: los parámetros se recogen en las variables `$1` a `$9`

Variable	Uso
`$0`	el nombre del script
`$1` a `$9`	parámetros del 1 al 9
`${10}`, `${11}`,...	parámetros por encima del 10
`$#`	número de parámetros
`$*`, `$@`	todos los parámetros

Ejemplo:

```
$ cat parms1.sh
#!/bin/bash
VAL=$((${1:-0} + ${2:-0} + ${3:-0}))
echo $VAL
$ bash parms1.sh 2 3 5
10
$ bash parms1.sh 2 3
5
```

shift

Desplaza los parámetros hacia la izquierda el número de posiciones indicado:

```
$ cat parms2.sh
#!/bin/bash
echo $#
echo $*
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}"
shift 9
echo $1 $2 $3
echo $#
echo $*
$ bash parms2.sh a b c d e f g h i j k l
12
a b c d e f g h i j k l
a b c d e f g h i j k
j k l
3
j k l
```

Diferencia entre `$*` y `$@`

Los parámetros `$*` y `$@` sólo se diferencian si van entrecomillados:

- `"$*"`
  se expande a una sola palabra, conteniendo todos los parámetros y con el valor de cada parámetro separado por el primer carácter de la variable especial `IFS` (por defecto, un espacio)
- `"$@"`
  cada parámetro se expande a una palabra separada; los parámetros entrecomillados se consideran uno solo aunque lleven espacios

Ejemplo:

```
$ cat parms3.sh
#!/bin/bash
IFS=":"
for par in "$*"
   do echo "Parámetro es:  $par"
done
echo "================================="
for par in "$@"
   do echo "Parámetro es:  $par"
done
$ bash parms3.sh hola "como estas hoy?" bien gracias
Parámetro es:  hola:como estas hoy?:bien:gracias
=================================
Parámetro es:  hola
Parámetro es:  como estas hoy?
Parámetro es:  bien
Parámetro es:  gracias
```

`set --`

El comando `set -- $var` copia las palabras de la variable `$var` en los parámetros posicionales `$1`, `$2`, ...

```
$ cat setdata
hola
como estás?
$
$ cat set.sh
#!/bin/bash
file=$(cat $1)   # Guarda en file el fichero pasado
set -- $file
echo $#
echo "$1 - $2 - $3"
$
$ bash set.sh setdata
3
hola - como - estás?
```

5.3. Entrada/salida

Es posible leer desde la entrada estándar o desde fichero usando `read` y redirecciones:

```
#!/bin/bash
echo -n "Introduce algo:  "
read x
echo "Has escrito $x"
echo -n "Escribe 2 palabras:  "
read x y
echo "Primera palabra $x; Segunda palabra $y"
```

Si queremos leer o escribir a un fichero utilizamos redirecciones:

```
echo $X > fichero
read X < fichero
```

Este último caso lee la primera línea de `fichero` y la guarda en la variable `X`

- Si queremos leer un fichero línea a línea podemos usar `while`:

```
#!/bin/bash
# FILE: linelist
# Usar:  linelist filein fileout
# Lee el fichero pasado en filein y
# lo salva en fileout con las lineas numeradas
count=0
while read BUFFER
do
   count=$((++count))
   echo "$count $BUFFER" » $2
done < $1
```

  - el fichero de entrada se va leyendo línea a línea y almacenando en BUFFER
  - `count` cuenta las líneas que se van leyendo

- El uso de lazos para leer ficheros es bastante ineficiente
  - deberían evitarse (por ejemplo, usar `cat fichero`)

Ejemplo de lectura de fichero

```
#!/bin/bash
# Usa $IFS para dividir la línea que se está leyendo
# por defecto, la separación es "espacio"
echo "Lista de todos los usuarios:"
OIFS=$IFS # Salva el valor de IFS
IFS=:  # /etc/passwd usa ":" para separar los campos
cat /etc/passwd |
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done
IFS=$OIFS # Recupera el $IFS original
```

- El fichero `/etc/passwd` se lee línea a línea
  - para cada línea, sus campos se almacenan en las variables que siguen a `read`
  - la separación entre campos la determina la variable `$IFS` (por defecto, espacio en blanco)

Redirecciones

Las redirecciones y pipes pueden usarse en otras estructuras de control

Ejemplo: lee las 2 primeras líneas de un fichero

```
if true
then
  read x
  read y
fi < fichero1
```

Ejemplo: lee líneas de teclado y guardalas en un fichero temporal convirtiendo minúsculas en mayúsculas

```
#/bin/bash
read buf
while [ "$buf" ]
do
  echo $buf
  read buf
done | tr 'a-z' 'A-Z' > tmp.$$
```

5.4. Tests

Los comandos que se ejecutan en un shell tienen un *código* de salida, que se almacena en la variable `$?`

- si `$?` es 0 el comando terminó bien
- si `$?` es > 0 el comando terminó mal

Ejemplo:

```
$ ls /bin/ls
/bin/ls
$ echo $?
0
$ ls /bin/ll
ls:  /bin/ll:  Non hai tal ficheiro ou directorio
$ echo $?
1
```

Podemos chequear la salida de dos comandos mediante los operadores `&&` (AND) y `||` (OR)

- estos operadores actúan en cortocircuito:

  comando1 `&&` *comando2*
  comando2 sólo se ejecuta si *comando1* acaba bien
  comando1 `||` *comando2*
  comando2 sólo se ejecuta si *comando1* falla

- comandos `true` y `false`: devuelven 0 y 1, respectivamente

Ejemplo con `&&`:

```
$ ls /bin/ls && ls /bin/ll
/bin/ls
ls:  /bin/ll:  Non hai tal ficheiro ou directorio
$ echo $?
1
$ ls /bin/ll && ls /bin/ls
ls:  /bin/ll:  Non hai tal ficheiro ou directorio
$ echo $?
1
```

Ejemplo con `||`:

```
$ ls /bin/ls || ls /bin/ll
/bin/ls
$ echo $?
0
$ ls /bin/ll || ls /bin/ls
ls:  /bin/ll:  Non hai tal ficheiro ou directorio
/bin/ls
$ echo $?
0
```

Estructura `if...then...else`

Podemos usar el estado de salida de uno o varios comandos para tomar decisiones:

```
if comando1
then
  ejecuta otros comandos
elif comando2
then
  ejecuta otros comandos
else
  ejecuta otros comandos
fi
```

- debe respetarse la colocación de los `then`, `else` y `fi`
  - también puede escribirse `if` *comando1* `; then`
- el `elif` y el `else` son opcionales, no así el `fi`

Ejemplo:

```
$ cat if.sh
#!/bin/bash
if (ls /bin/ls && ls /bin/ll) >/dev/null 2>&1
then
  echo "Encontrados ls y ll"
else
  echo "Falta uno de los ficheros"
fi
$ bash if.sh
Falta uno de los ficheros
```

Comando `test`

Notar que `if` sólo chequea el código de salida de un comando, no puede usarse para comparar valores: para eso se usa el comando `test`
El comando test permite:

- chequear la longitud de un string
- comparar dos strings o dos números
- chequear el tipo de un fichero
- chequear los permisos de un fichero
- combinar condiciones juntas

`test` puede usarse de dos formas:

`test` *expresión*

o bien

`[` *expresión* `]`[4]

Si la expresión es correcta `test` devuelve un código de salida 0, si es falsa, devuelve 1:

- este código puede usarse para tomar decisiones:

```
if [ "$1" = "hola" ]
then
  echo "Hola a ti también"
else
  echo "No te digo hola"
fi
if [ $2 ]
then
  echo "El segundo parámetro es $2"
else
  echo "No hay segundo parámetro"
fi
```

- en el segundo `if` la expresión es correcta si `$2` tiene algún valor; falsa si la variable no está definida o contiene null (`""`)

[4]Notar los espacios en blanco entre los [ ] y *expresión*

Expresiones

Existen expresiones para chequear strings, números o ficheros

Chequeo de strings

Expresión	Verdadero sí
string	el string es no nulo ("")
`-z` *string*	la longitud del string es 0
`-n` *string*	la longitud del string no es 0
string1 `=` *string2*	los strings son iguales
string1 `!=` *string2*	los strings son distintos

Chequeo de enteros

Expresión	Verdadero sí
int1 `-eq` *int2*	los enteros son iguales
int1 `-ne` *int2*	los enteros son distintos
int1 `-gt` *int2*	*int1* mayor que *int2*
int1 `-ge` *int2*	*int1* mayor o igual que *int2*
int1 `-lt` *int2*	*int1* menor que *int2*
int1 `-le` *int2*	*int1* menor o igual que *int2*

Chequeo de ficheros

Expresión	Verdadero sí
`-e` *file*	*file* existe
`-r` *file*	*file* existe y es legible
`-w` *file*	*file* existe y se puede escribir
`-x` *file*	*file* existe y es ejecutable
`-f` *file*	*file* existe y es de tipo regular
`-d` *file*	*file* existe y es un directorio
`-c` *file*	*file* existe y es un dispositivo de caracteres
`-b` *file*	*file* existe y es un dispositivo de bloques
`-p` *file*	*file* existe y es un pipe
`-S` *file*	*file* existe y es un socket
`-L` *file*	*file* existe y es un enlace simbólico
`-u` *file*	*file* existe y es *setuid*
`-g` *file*	*file* existe y es *setgid*
`-k` *file*	*file* existe y tiene activo el *sticky bit*
`-s` *file*	*file* existe y tiene tamaño mayor que 0

Operadores lógicos con `test`

Expresión	Propósito
`!`	invierte el resultado de una expresión
`-a`	operador AND
`-o`	operador OR
`(` *expr* `)`	agrupación de expresiones; los paréntesis tienen un significado especial para el shell, por lo que hay que *escaparlos*

Ejemplos:

```
$ test -f /bin/ls -a -f /bin/ll ; echo $?
1
$ test -c /dev/null ; echo $?
0
$ [ -s /dev/null ] ; echo $?
1
$ [ ! -w /etc/passwd ] && echo "No puedo escribir"
No puedo escribir
$ [ $$ -gt 0 -a \( $$ -lt 5000 -o -w file \) ]
```

Comando de test extendido

A partir de la versión 2.02 de Bash se introduce el *extended test command*: `[[ expr ]]`

- permite realizar comparaciones de un modo similar al de lenguajes estándar:
  - permite usar los operadores && y || para unir expresiones
  - no necesita *escapar* los paréntesis

Ejemplos:

```
$ [[ -f /bin/ls && -f /bin/ll ]] ; echo $?
1
$ [[ $$ -gt 0 && ($$ -lt 5000 || -w file) ]]
```

5.5. Control de flujo

Además del `if` bash permite otras estructuras de control de flujo: `case`, `for`, `while` y `until`

Estructura `case`

Formato:

```
case valor in
  patrón_1)
    comandos si value = patrón_1
    comandos si value = patrón_1 ;;
  patrón_2)
    comandos si value = patrón_2 ;;
  *)
    comandos por defecto ;;
esac
```

- si *valor* no coincide con ningún patrón se ejecutan los comandos después del `*)`
  - esta entrada es opcional
- *patrón* puede incluir comodines y usar el símbolo | como operador OR

Ejemplo:

```
#!/bin/bash
echo -n "Respuesta:  "
read RESPUESTA
case $RESPUESTA in
  S* | s*)
    RESPUESTA="SI" ;;
  N* | n*)
    RESPUESTA="NO" ;;
  *)
    RESPUESTA="PUEDE" ;;
esac
echo $RESPUESTA
```

Lazos `for`

Formato:

```
for var in lista
do
  comandos
done
```

- *`var`* toma los valores de la lista
  - puede usarse *globbing* para recorrer los ficheros

Ejemplo: recorrer una lista

```
LISTA="10 9 8 7 6 5 4 3 2 1"
for var in $LISTA
do
  echo $var
done
```

Ejemplo: recorrer los ficheros `*.bak` de un directorio

```
dir="/var/tmp"
for file in $dir/*.bak
do
  rm -f $file
done
```

Sintaxis alternativa, similar a la de C

```
LIMIT=10
for ((a=1, b=LIMIT; a <= LIMIT; a++, b--))
do
  echo "$a-$b"
done
```

Bucle `while`

Formato:

```
while comando
do
  comandos
done
```

- se ejecuta mientras que el código de salida de `comando` sea cierto

Ejemplo:

```
while [ $1 ]
do
  echo $1
  shift
done
```

Bucle `until`

Formato:

```
until comando
do
  comandos
done
```

- se ejecuta hasta que el código de salida de `comando` sea hace cierto

Ejemplo:

```
until [ "$1" = "" ]
do
  echo $1
  shift
done
```

`break` y `continue`

Permiten salir de un lazo (`break`) o saltar a la siguiente iteración (`continue`)

- `break` permite especificar el número de lazos de los que queremos salir (`break` *n*)

Ejemplo con `break`:

```
# Imprime el contenido de los ficheros hasta que
# encuentra una línea en blanco
for file in $*
do
  while read buf
  do
    if [ -z "$buf" ]
    then
      break 2
    fi
    echo $buf
  done < $file
done
```

Ejemplo con `continue`:

```
# Muestra un fichero pero no las líneas de más
# de 80 caracteres
while read buf
do
  cuenta=`echo $buf | wc -c`
  if [ $cuenta -gt 80 ]
  then
    continue
  fi
  echo $buf
done < $1
```

5.6. Funciones

Podemos definir funciones en un script de shell:

```
funcion() {
  comandos
}
```

y para llamarla:

```
funcion p1 p2 p3
```

Siempre tenemos que definir la función antes de llamarla:

```
#!/bin/bash
# Definición de funciones
funcion1() {
  comandos
}
funcion2() {
  comandos
}
# Programa principal
funcion1 p1 p2 p3
```

Paso de parámetros

La función referencia los parámetros pasados por posición, es decir, `$1`, `$2`, ..., y `$*` para la lista completa:

```
$ cat funcion1.sh
#!/bin/bash
funcion1()
{
  echo "Parámetros pasados a la función:  $*"
  echo "Parámetro 1:  $1"
  echo "Parámetro 2:  $2"
}
# Programa principal
funcion1 "hola" "que tal estás" adios
$
$ bash funcion1.sh
Parámetros pasados a la función:  hola que tal estás adios
Parámetro 1:  hola
Parámetro 2:  que tal estás
```

Variables locales

Es posible definir variables locales en las funciones:

```
$ cat locales.sh
#!/bin/bash
testvars() {
  local localX="localX en función"
  X="X en función"
  echo "Dentro de la función:  $localX, $X, $globalX"
}
# Programa principal
localX="localX en main"
X="X en main"
globalX="globalX en main"
echo "Dentro de main:  $localX, $X, $globalX"
# Llama a la función
testvars
echo "Otra vez dentro de main:  $localX, $X, $globalX"

$ bash locales.sh
Dentro de main:  localX en main, X en main, globalX en
main
Dentro de la función:  localX en función, X en función,
globalX en main
Otra vez dentro de main:  localX en main, X en función,
globalX en main
```

`return`

Después de llamar a una función, `$?` tiene el código se salida del último comando ejecutado:

- podemos ponerlo de forma explícita usando `return`

```
#!/bin/bash
funcion2() {
  if [ -f /bin/ls -a -f /bin/ln ]; then
    return 0
  else
    return 1
  fi
}
# Programa principal
if funcion2; then
  echo "Los dos ficheros existen"
```

```
else
  echo "Falta uno de los ficheros - adiós"
  exit 1
fi
```

`return` permite devolver un entero entre 0 y 255

```
#!/bin/bash
# maximum.sh:  Máximo de dos enteros
EQUAL=0
max() {
  if [ "$1" -eq "$2" ]; then
    return $EQUAL
  elif [ "$1" -gt "$2" ]; then
    return $1
  else
    return $2
  fi }
max 36 34
salida=$?
if [ "$salida" -eq $EQUAL ]; then
  echo "Los números son iguales"
else
  echo "El mayor es $salida"
fi
```

5.7. Arrays

Bash soporta arrays unidimensionales

- No es necesario especificar su tamaño ni reservar memoria
- Los índices empiezan a contar en cero
- Los elementos de un array pueden ser de tipos diferentes
- Formas de inicialización:
  - `array[`*índice*`]=`*valor*
  - `array=(`*valor0 valor1 ...*`)`
  - `array=([`*índice*`]=`*valor* `[`*índice*`]=`*valor ...*`)`
  - `declare -a array`

- Para acceder a un elemento de un array tenemos que ponerlo entre llaves: `${array[`*índice*`]}`

Ejemplo de uso de arrays:

```
$ cat array1.sh
#!/bin/bash
# Los elementos del array no necesitan ser consecutivos
array[11]=23
array[13]=37
array[5]=$((${array[11]}+ ${array[13]}))
echo "array[5]=${array[5]}"
# Otra forma de inicializar un array
array2=( cero uno dos tres cuatro )
echo "array2[3]=${array2[3]}"
echo "Elementos en array2=${#array2[@]}"
# Una tercera forma
array3=( [8]=ocho [10]=10 [13]=trece )
echo "array3[10]=${array3[10]}"
echo "array3=${array3[@]}"
echo "array3=${array3[@]:10}"
```

```
$ bash array1.sh
array[5]=60
array2[3]=tres
Elementos en array2=5
array3[10]=10
array3=ocho 10 trece
array3=10 trece
```

5.8. Otros comandos

`wait`

Permite esperar a que un proceso lanzado en *background* termine

```
sort $largefile > $newfile &
ejecuta comandos
wait
usa $newfile
```

Si lanzamos varios procesos en background podemos usar `$!`

- `$!` devuelve el `PID` del último proceso lanzado

```
sort $largefile1 > $newfile1 &
SortPID1=$!
sort $largefile2 > $newfile2 &
SortPID2=$!
ejecuta comandos
wait $SortPID1
usa $newfile1
wait $SortPID2
usa $newfile2
```

`trap`

Permite *atrapar* las señales del sistema operativo

- permite hacer que el programa termine limpiamente (p.e. borrando ficheros temporales, etc.) aún en el evento de un error

```
$ cat trap.sh
#!/bin/bash
cachado() {
  echo "Me has matado!!!"
  kill -15 $$
}
trap "cachado" 2 3
while true; do
  true
done
$ bash trap.sh
(Ctrl-C)
Me has matado!!!
Terminado
```

Las señales más comunes para usar con `trap` son:

Señal	**Significado**
0	salida del shell (por cualquier razón, incluido fin de fichero)
1	colgar
2	interrupción (Ctrl-C)
3	quit
9	kill (no puede ser parada ni ignorada)
15	terminate; señal por defecto generada por kill

`$RANDOM`

`$RANDOM` es una función interna de bash (no una constante) que devuelve un entero pseudoaleatorio entre 0-32676

```
$ echo $RANDOM
3487
$ echo $RANDOM
27932
```

`printf`

Puede usarse en sustitución de `echo`; sintaxis similar a C

```
$ printf "%d\n" 5
5
$ printf "%f\n" 5
5.000000
```

`exit`

Finaliza el script

- se le puede dar un argumento numérico que toma como estado de salida, p.e. `exit 0` si el script acaba bien y `exit 1` en caso contrario
- si no se usa `exit`, el estado de salida del script es el del último comando ejecutado

`exec`

`exec` ejecuta un comando reemplazando al shell actual

```
#!/bin/bash
echo "Antes del exec"
exec comando
echo "Después del exec"
```

el segundo `echo` no se ejecuta, y *`comando`* se ejecuta con el PID con que se lanzó el script

5.9. Optimización y depuración de scripts

El shell no es especialmente eficiente a la hora de ejecutar trabajos pesados

- Ejemplo: script que cuenta las líneas de un fichero:

```
$ cat cuentalineas1.sh
#!/bin/bash
count=0
while read line
do
  count=$(expr $count + 1)
done < $1
echo "El fichero $1 tiene $count líneas"
```

  - si medimos el tiempo que tarda

```
$ time bash cuentalineas1.sh Quijote.txt
El fichero Quijote.txt tiene 36855 líneas
real 0m59.757s
user 0m17.868s
sys  0m41.462s
```

- Podemos mejorarlo si usamos aritmética de shell en vez de el comando `expr`

```
$ cat cuentalineas2.sh
#!/bin/bash
count=0
while read line
do
  count=$(($count+1))
done < $1
echo "El fichero $1 tiene $count líneas"
```

  - el tiempo ahora

```
$ time bash cuentalineas2.sh Quijote.txt
El fichero Quijote.txt tiene 36855 líneas
real 0m1.014s
user 0m0.887s
sys  0m0.108s
```

- Y todavía mejor:

```
$ cat cuentalineas3.sh
#!/bin/bash
count=$(wc -l $1 | cut -d " " -f 1)
echo "El fichero $1 tiene $count líneas"
$
$ time bash cuentalineas3.sh Quijote.txt
El fichero Quijote.txt tiene 36855 líneas
real 0m0.096s
user 0m0.005s
sys  0m0.009s
```

- Conclusiones
  - Intenta reducir el número de procesos creados al ejecutar el script, por ejemplo, usando las funciones aritméticas del shell
  - Siempre que sea posible, intenta usar comandos del shell (`wc`, `tr`, `grep`, `sed`, etc.) en vez de lazos

Depuración

Para depurar un script de shell podemos usar la opción `-x` o `-o xtrace` de bash:

- muestra en la salida estándar trazas de cada comando y sus argumentos, después de que el comando se haya expandido pero antes de que se sea ejecutado

```
$ bash -x cuentalineas3.sh Quijote.txt
++ wc -l Quijote.txt
++ cut -d ' ' -f 1
+ count=36855
+ echo 'El fichero Quijote.txt tiene 36855 líneas'
El fichero Quijote.txt tiene 36855 líneas
```

- La opción `-v` o `-o verbose` muestra cada línea del script antes de que se expanda

```
$ bash -v cuentalineas3.sh Quijote.txt
#!/bin/bash
count=$(wc -l $1 | cut -d " " -f 1)
wc -l $1 | cut -d " " -f 1
```

```
echo "El fichero $1 tiene $count líneas"
El fichero Quijote.txt tiene 36855 líneas
```

- Ambas opciones pueden combinarse

```
bash -xv cuentalineas3.sh Quijote.txt
#!/bin/bash
count=$(wc -l $1 | cut -d " " -f 1)
wc -l $1 | cut -d " " -f 1
++ wc -l Quijote.txt
++ cut -d ' ' -f 1
+ count=36855
echo "El fichero $1 tiene $count líneas"
+ echo 'El fichero Quijote.txt tiene 36855 líneas'
El fichero Quijote.txt tiene 36855 líneas
```

Es posible depurar sólo parte de un script:

- poner `set -x` o `set -xv` al inicio del trozo a depurar
- `set +x` o `set +xv` para cancelar

```
$ cat cuentalineas3.sh
#!/bin/bash
set -x
count=$(wc -l $1 | cut -d " " -f 1)
set +x
echo "El fichero $1 tiene $count líneas"
$
$ bash cuentalineas3.sh Quijote.txt
++ wc -l Quijote.txt
++ cut -d ' ' -f 1
+ count=36855
+ set +x
El fichero Quijote.txt tiene 36855 líneas
```

5.10. Prácticas: Programación shell-script

Obligatorio el uso de `vi` para desarrollar todos los scripts. Los scripts deben de estar adecuadamente comentados.

1. Escribir un script que reciba un nombre de archivo, pedido al usuario explícitamente con el comando `read`, verifique que existe y que tiene permiso de lectura para el propietario, fallando en caso contrario con

un mensaje de error y saliendo del script con un código de salida 1. Si tiene permiso, debe añadirle permisos de ejecución para el propietario y el grupo. Por último, debe mostrar cómo quedan los permisos (usando `ls -l` y el comando `cut` para que se vean solo los permisos) y ejecutar el fichero (usando `exec`).

2. Crear un script que recorra los archivos con extensión .txt y .c del directorio actual y los coloque en carpetas por fecha de última modificación del fichero. Para ello, debe obtener la fecha de última modificación del fichero usando el comando `stat` y crear un directorio, dentro del actual, cuyo nombre siga el formato *año*`/`*mes*`/`*día*, obteniendo estos valores de la fecha de última modificación del fichero, por ejemplo `2018/12/31`. Cada fichero debe moverse al directorio correspondiente a su fecha. Adicionalmente, al terminar el script debe indicar el número de ficheros de cada tipo que se movieron. Requisitos:

   (a) El script no debe crear ficheros nuevos ni modificar los existentes. Podéis usad el comando `touch` (fuera del script) para crear ficheros y especificarles una fecha de última modificación concreta.

   (b) Antes de mover un archivo, el script debe comprobar de que se trata de un fichero regular y que tiene permisos de lectura sobre el mismo. Si no se cumple, debe imprimir un mensaje de aviso y continuar.

   (c) La obtención del la fecha del fichero y la creación del directorio debe de hacerse **en una función**.

3. Programar un script que al pasarle como parámetro en la línea de comandos el nombre de un archivo de texto, ordene las líneas del texto por orden alfabético, ascendentemente si no se le pasa un segundo parámetro o si este es una `a`, o descendentemente si se le pasa una `z`, guardando la versión ordenada en otro archivo, en el directorio actual, con el mismo nombre que el original pero precedido de `ordenado_`. Al finalizar, el script debe indicar el número de líneas del fichero. Requisitos:

   (a) El script debe comprobar que se le pasan 1 o 2 parámetros, y en caso contrario, escribir un mensaje con instrucciones de uso y salir del script con un código de salida 1.

   (b) Debe comprobar que el fichero es regular y que tiene permiso de lectura, fallando en caso contrario, escribiendo un mensaje de aviso y saliendo del script con un código de salida 1.

(c) Debe también comprobar que el segundo parámetro es una `a` o una `z`, fallando en caso contrario, escribiendo un mensaje de aviso y saliendo del script con un código de salida 1.

4. VOLUNTARIO: Añade dos nuevas opciones al script anterior para que ordene aleatoriamente las líneas del texto si se pasa como parámetro una `r`, y en orden inverso al pasarle una `i`.

5. VOLUNTARIO: Escribir un script que implemente un comando de borrado seguro. Los ficheros que se pasan como argumento a ese script se copiarán al directorio denominado `.basura` del directorio home del usuario. En cada invocación del script, adicionalmente se borrarán todos los ficheros que tengan más de 72 horas en `.basura`. Además, escribir un segundo script que recupere los archivos en `.basura` y los deje con su nombre original y en el directorio en el que se encontraban antes de ser borrados (si ese directorio existe, en caso de que no exista debe de recuperarse el fichero al directorio actual). Debe preguntarse al usuario explícitamente qué ficheros se recuperan de todos los guardados.

   Entrega y plazo: Empaquetar los scripts (los tres obligatorios por un lado y los voluntarios por otro) en archivos tar.gz. Los script deben funcionar en un sistema Linux (tened cuidado si usáis Windows para editar los scripts, o para ponerle los comentarios, ya que los retornos de carro son diferentes en Linux y Windows). Se pueden remitir a través del campus virtual hasta el 3 de marzo los obligatorios y hasta el 17 de marzo los voluntarias.