BASH(1) General Commands Manual BASH(1)

NAME

bash - GNU Bourne-Again SHell

Sistemas Operativos II

Programación Shell-Script

Bash is intended to be a conformant implementation of the Shell and Utilities portion of the IEEE POSIX specification (IEEE Standard 1003.1). **Bash** can be configured to be POSIX-conformant by default.

OPTIONS

All of the single-character shell options documented in the description of the **set** builtin command can be used as options when the shell is invoked. In addition, **bash** interprets the following options when it is invoked:

-c If the **-c** option is present, then commands are read from the first non-option argument command_string. If there are arguments after the command_string, the first argument is assigned to **$0** and any remaining arguments are assigned to the positional parameters. The assignment to **$0** sets the name of the shell, which is used in warning and error messages.

-i If the **-i** option is present, the shell is interactive.

-l Make **bash** act as if it had been invoked as a login shell (see **INVOCATION** below).

-r If the **-r** option is present, the shell becomes restricted (see **RESTRICTED SHELL** below).

-s If the **-s** option is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell.

-v Print shell input lines as they are read.

-x Print commands and their arguments as they are executed.

-D A list of all double-quoted strings preceded by **$** is printed on the standard output. These are the strings that are subject to language translation when the current locale is not **C** or **POSIX**. This implies the **-n** option; no commands will be executed.

[-+]O [shopt_option]

shopt_option is one of the shell options accepted by the **shopt** builtin (see **SHELL BUILTIN COMMANDS** below). If shopt_option is present, **-O** sets the value of that option; **+O** unsets it. If shopt_option is not supplied, the names and values of the shell options accepted by **shopt** are printed on the standard output. If the invocation option is **+O**, the output is displayed in a format that may be reused as input.

-- A -- signals the end of options and disables further option processing. Any arguments after the -- are treated as file-

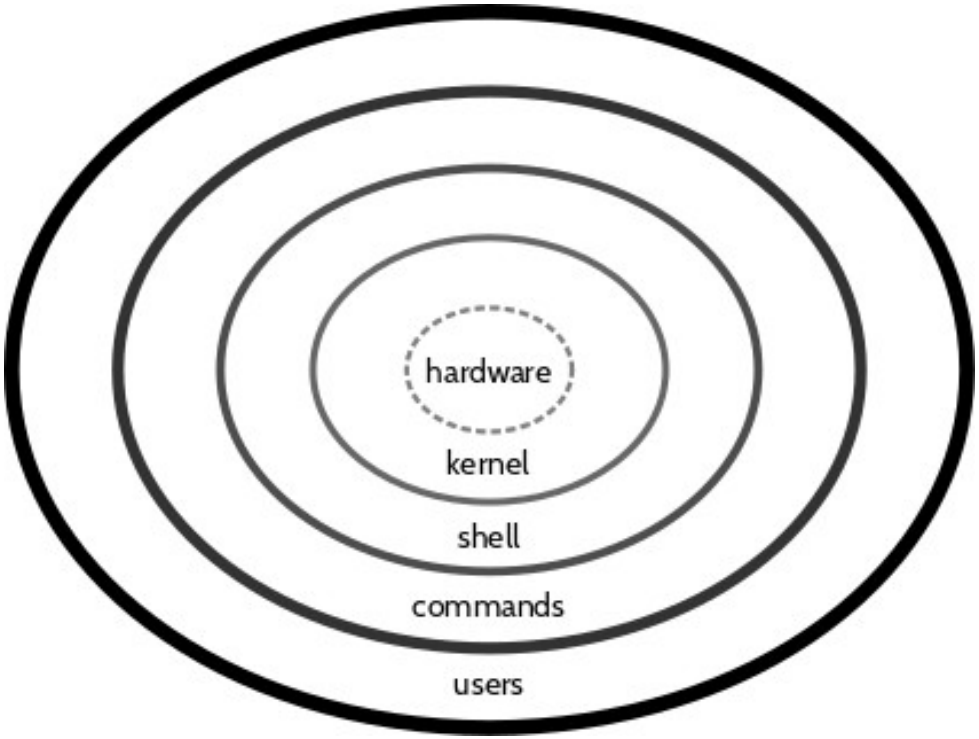
Manual page bash(1) line 1 (press h for help or q to quit)

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

CAMPUS VIDA CAMPUS DE EXCELENCIA INTERNACIONAL

- El Shell (intérprete de comandos)
  - **Interfaz** de línea de comandos
    - Comunicación directa con el **Sistema Operativo**
    - Traduce líneas de comandos en instrucciones del sistema operativo
- Programación **Shell-Script**
  - Secuencia de **comandos de shell**
    - Administración del sistema, programación de tareas repetitivas
  - Lo que aprendas del **uso de comandos** desde la interfaz te servirá en la programación de **shell-scripts**

- El Shell (intérprete de comandos)
  - En UNIX no está integrado en el kernel
    - Puedes escribir tu propio intérprete de comandos



- Principales shells
  - `sh` (Bourne Shell)
    - Primer shell en las primeras versiones de UNIX
  - `csh` (C Shell)
    - Creado por Bill Joy (vi editor)
    - Desarrollado para el sistema UNIX BSD de Berkeley
  - `tcsh` (Turno C Shell)
    - Versión mejorada de `csh`
  - `ksh` (Korn Shell)
    - Basado en `sh` (Bourne Shell) con características del `csh` (C Shell)

- Principales shells
  - **`bash`** (Bourne Again Shell)
    - Shell por defecto en Linux
    - Tiene todas las características de `bourne shell`
    - Incluye características avanzadas de C
- Usaremos `bash` en las **prácticas**
  - La sintaxis es muy parecida entre diferentes shells
  - Scripts en `sh` se ejecutará en shells derivados como el `bash`
- Otros shells: fish, ash, powershell, etc
  - Comparativa: https://en.wikipedia.org/wiki/Comparison_of_command_shells

- Funcionamiento del shell

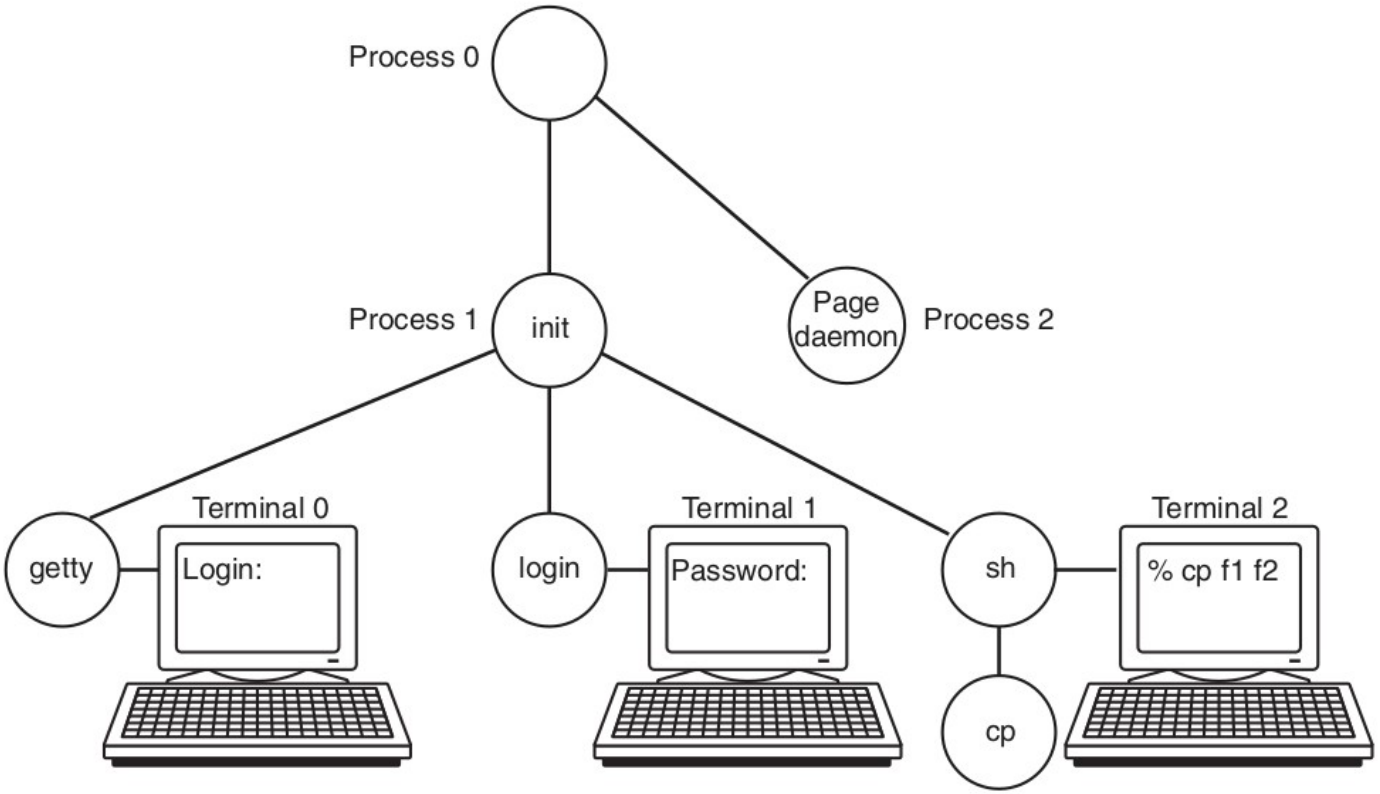
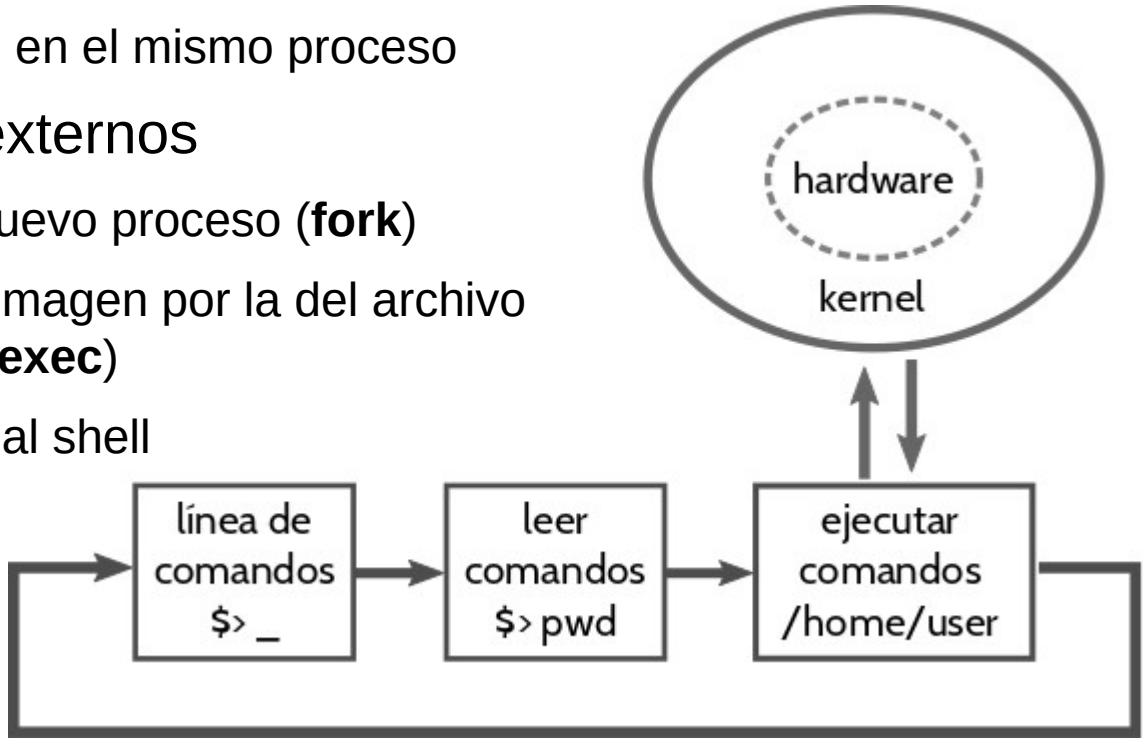


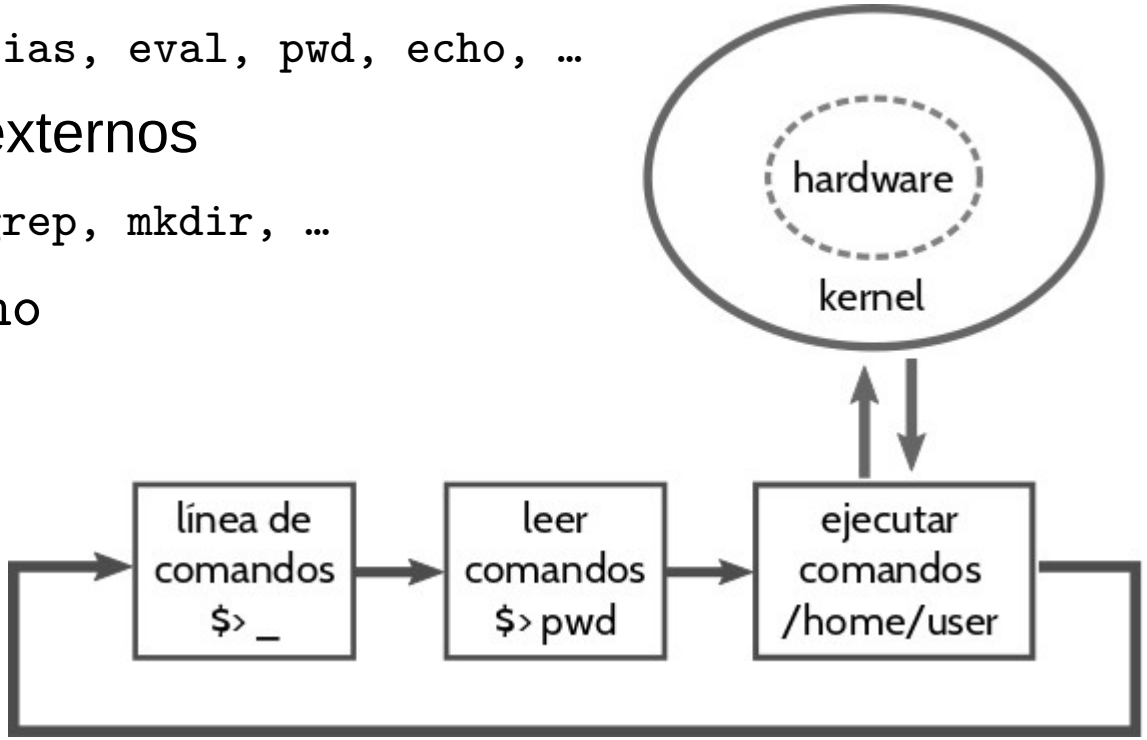
Figure 10-11. The sequence of processes used to boot some Linux systems.

Sistemas operativos modernos - Tanenbaum - 4ed (2016)

- Funcionamiento del shell
  - El shell es un **proceso iterativo** que ejecuta comandos
  - Comandos internos al shell
    - Se ejecutan en el mismo proceso
  - Comandos externos
    - Inician un nuevo proceso (**fork**)
    - Cambia su imagen por la del archivo ejecutable (**exec**)
    - Son ajenos al shell



- Funcionamiento del shell
  - El shell es un **proceso iterativo** que ejecuta comandos
  - Comandos internos al shell
    - `cd, bg, alias, eval, pwd, echo, …`
  - Comandos externos
    - `cp, cat, grep, mkdir, …`
  - `$> type echo`



- Funcionamiento del shell
  - El shell es un **proceso iterativo** que ejecuta comandos

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, parameters);      /* read input from terminal */

        if (fork( ) != 0) {                     /* fork off child process */
             /* Parent code. */
             waitpid(–1, &status, 0);           /* wait for child to exit */
        } else {
             /* Child code. */
             execve(command, parameters, 0);    /* execute command */
        }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1. Sistemas operativos modernos - Tanenbaum - 4ed (2016)

- Línea de comandos `$>_`
  - Los comandos usualmente constan de 3 **componentes**
    - `COMANDO [OPCIONES] [PARAMETROS]`
    - `$> ls`
    - `$> ls -l`
    - `$> ls -l /tmp`
  - **Ejecución** de comandos
    - Los comandos externos se buscan en los directorios indicados en la variable PATH
    - `/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin`

- Línea de comandos `$>_`
  - Ejecución en primer plano (**foreground**)
    - El shell espera (*waitpid*) a que termine el comando antes de aceptar uno nuevo
  - Ejecución en segundo plano (**background**)
    - `$> sleep 10 &`
    - `$> _` (el shell regresa a línea de comandos)
  - Pausar un comando
    - CTRL-Z
  - Terminar un comando
    - CTRL-C

- Línea de comandos $>_
  - Ver la lista de comandos en background (**jobs**)
    - $> sleep 20 &
    - $> sleep 50 (pulsar CTRL-Z)
    - $> jobs

```
Job  Group  CPU   State     Command
1    6311   0%    running   sleep 20 &
2    6414   0%    stopped   sleep 50
```

  - Traer un comando a primer plano
    - **fg** %job_id
  - Llevar un proceso a segundo plano
    - **bg** %job_id

- Línea de comandos `$>_`
  - Ejecutar varios **comandos seguidos**
    - `$> cd /tmp; sleep 10; ls -l`
  - **Completar** nombre del comando pulsando TAB
    - `$> ifc (pulsar TAB)` → `$> ifc`**`onfig`**
    - Muy útil para completar nombres de ficheros y directorios
  - Podemos **movernos** por los comandos, opciones y argumentos
    - `CTRL-FLECHA, <INICIO>, <FIN>, ...`
    - Mejora la productividad

- Línea de comandos `$>_`
  - **Histórico** de comandos
    - Listado de comandos previamente usados
    - `$> history`

```
...
1998  cd /tmp; sleep 10; ls -l
1999  ifconfig
2000  history
```

    - Podemos **volver a ejecutar** un comando
      - `<FLECHA-ARRIBA>, <FLECHA-ABAJO>`
      - `!n`
      - `Ctrl-R` búsqueda inversa
      - `fc [primero] [último]` repite varios comandos del historial

- Línea de comandos `$>_`
  - Páginas de **manual**
    - Información concreta sobre la sintaxis y opciones
    - `$> man history`
    - Para comandos internos de **bash**: `help`
  - Páginas de **información**
    - Sistema de manuales del proyecto GNU
    - Más flexible y completo que `man`
    - Proporciona información basada en hipertexto
    - `$> info history`
  - Información sobre **aplicaciones**
    - Ver directorios `/usr/doc` o `/usr/share/doc`

- Línea de comandos `$>_`
  - `COMANDO [OPCIONES] [PARAMETROS]`
  - **Comodines** (*wildcards*)
    - **Expansión** del shell o *globbing*
    - Permite especificar varios **ficheros** como parámetros
    - `$> ls *.log`
    - `$> ls /tmp/[A-Za-z]*`

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

- Línea de comandos $>_
  - Caracteres **especiales**
    - &
    - *, ?, [ ], [!]
    - $
    - ;
    - <, >, <<, >>, `, |
  - El shell los reconoce y los trata de forma especial
    - Globbing, ejecución, variables, redirecciones, ...

- Línea de comandos `$>_`
  - **Eliminar significado especial**
  - `'` (comilla simple)
    - Ignora **todos** los caracteres especiales
  - `"` (comilla doble)
    - Ignora **todos** los caracteres especiales **excepto** `$`, `\`, `` ` ``
  - `\` (barra invertida)
    - Ignora el **carácter** especial que sigue a `\`

- Línea de comandos $>_
  - Otras **expansiones**
    - Además de las ya vistas para ficheros

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

    - bash permite otras expansiones para **parámetros**
      - Expansión de llaves
      - Expansión de tilde
      - Expansión aritmética

- Línea de comandos $>_
  - Otras **expansiones**
    - Expansión de **llaves** {} para generación de strings
      - `$> echo a{1,2,3}b`
        `a1b a2b a3b`
    - Expansión de **tilde** ~ para directorio de usuario
      - `$> cat ~/.bash_history`
    - Expansión **aritmética** `$(( expr ))` o `$[ expr ]` para evaluar expresiones
      - `expr` tiene una sintaxis similar al lenguaje C
      - `$> echo $(((4+11)/3))`
      - `$> echo $[(4+11)/3]`

- Línea de comandos `$>_`
  - **Variables** de shell
    - Podemos **crear variables** desde la linea de comandos

```
$> texto='hola'
$> i=7
$> let j=i+1
$> k=$((j/2))
```

  - Importante! **sin espacios** antes ni después del =
  - También podemos crear variables usando el `comando` **`read`**
    - `$> read texto (PULSAR ENTER)`
    - Escribir contenido que queremos asignar y pulsar enter
  - Vaciar una variable (valor **`NULL`**)

```
$> texto=""
$> i=
```

- Línea de comandos `$>_`
  - Acceder al **contenido** de una variable usando **`$`**`variable`
    - `$> echo $texto`
    - `$> echo $i`
  - `Comando` **`echo`**
    - Muestra una línea de texto
    - `$> echo “hola mundo”`
  - Podemos **combinar texto** con variables
    - `$> echo "El usuario es: $USER"`
    - `$> echo 'El usuario es: $USER'`
    - RECUERDA: significado de las comillas simples y dobles

- Línea de comandos `$>_`
  - Variables **locales**
    - Visibles sólo desde el shell actual
    - Preferiblemente en minúsculas
  - Variables de **entorno**
    - `export variable` para ser visible en todo los shells hijos (fork)
    - En mayúsculas siempre
    - `$> HISTFILE=/home/pablo/.bash_history`
    - `$> export HISTFILE`
  - Ver **listado** de variables
    - `$> printenv`

- Línea de comandos $>_
  - (algunas) variables de **entorno**

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

- Línea de comandos `$>_`
  - **Evaluación** de comandos (*no confundir con comando test*)
    - Los comandos tienen un **código de salida** que se almacena en **`$?`**
      - Si `$? = 0` el comando terminó bien
      - Si `$? > 0` el comando terminó con un error
    - `$> ls -l /tmp/unknown_file; echo $?`
  - Ejecutar varios **comandos seguidos** condicionalmente
    - **`&&`** Operador **AND**
      - `$> cd /tmp && ls -l`
    - **`||`** Operador **OR**
      - `$> cd /tmpo || echo 'Error $?: Directorio no disponible'`

- Línea de comandos $>_
  - **Redirección** de entrada salida (E/S)
    - Toda la E/S se realiza a través de **ficheros**
    - Cada proceso tiene asociado 3 ficheros de E/S
      - Descriptor Nombre Destino
        0 entrada estándar (stdin) teclado
        1 salida estándar (stdout) pantalla
        2 error estándar (stderr) pantalla
  - Podemos redireccionar la E/S **usando** <, >, <<, >>, |
    - `$> ls -l > listado`
    - `$> ls -l > /dev/lp`
    - `$> cat < listado`
    - `$> ls -l >> listado`

- Línea de comandos `$>_`
  - Redirección el **error estándar** (descriptor 2) al fichero `error.log`
    - `$> ls nofile 2> error.log`
  - Redirección la **salida estándar** (descriptor 1) y el **error estándar** (descriptor 2) al fichero `dirlist.log`
    - `$> ls > dirlist.log 2>&1`
  - Canalización (`PIPE`)
    - Redirección de la **salida** de un comando a la **entrada** del siguiente
    - `$> ls -l | wc -l`
    - `$> sort < certificate.pem |tail -5`

- Línea de comandos `$>_`
  - **Orden** de **evaluación** al ejecutar comandos
    - 1. Redirección de E/S
      - `<, >, <<, >>, |`
    - 2. Expansión de variables
      - `$USER → pablo`
    - 3. Expansión de nombres de fichero
      - Sustituye comodines por los nombres de fichero
      - `ls *.log → error.log system.log`
  - ¿Por qué **`ls | more`** funciona y **`ls $pipe more`** no?
    - `$> ls | more`
    - `$> pipe=\|`
    - `$> ls $pipe more`
      - `ls: no se puede acceder a '|': No existe el archivo o el directorio`
      - `ls: no se puede acceder a 'more': No existe el archivo o el directorio`

- Línea de comandos `$>_`
  - Comandos para procesamiento de texto (**`FILTROS`**)
  - `cat, head, tail, grep, find, sort, …`
    - Permiten mostrar, buscar, ordenar, formatear ficheros
    - Usan la **entrada estándar** (o un fichero)
    - Envían la salida a la **salida estándar**
    - Se pueden **canalizar** y **redireccionar**
  - `Evitar` **`Useless Use of Cat`**
  - Puedes **ver un resumen** en la sección 4.1 del material de apoyo al **Tema 5: Uso del shell** en el campus virtual
    - Estudia las opciones de cada comando: `$> man comando`