

# UNIX, *Shell* y *Scripts*

Francisco Rosales García  
Ángel Herranz

Universidad Politécnica de Madrid

Otoño 2012



# Índice general

<b>1. El Entorno UNIX</b>	<b>1</b>
1.1. ¿Qué es UNIX?	1
1.2. Usuarios y Grupos	3
1.3. Sesión	3
1.4. Mandatos	4
1.5. Procesos	7
1.6. Árbol de Ficheros	7
1.7. Descriptores de fichero	11
1.8. Intérprete de mandatos	12
1.9. Variables de entorno	13
1.10. Ficheros de texto ejecutables	14
<b>2. Uso del <i>Shell</i></b>	<b>15</b>
2.1. El <i>prompt</i>	15
2.2. Estado de terminación	15
2.3. Primer y segundo plano	16
2.4. Redirección	16
2.5. Secuencia de mandatos	17
2.6. Metacaracteres	18
2.7. Uso interactivo	19
2.8. Algunos mandatos útiles	22
2.9. Configuración	25
<b>3. Programación de <i>Scripts</i></b>	<b>27</b>
3.1. Edición	28
3.2. Comentarios	28
3.3. Variables y Entorno	28
3.3.1. Variable PATH	29
3.3.2. Variables especiales	29
3.4. El mandato <code>test</code>	30
3.5. Funciones	32
3.6. Entrada, salida y error estándar	33
3.7. Manipulación de tiras de caracteres	34



# Capítulo 1

## El Entorno UNIX

En este apartado intentaremos hacernos una idea de cuáles son las características y los conceptos básicos que se manejan en el entorno UNIX.

### 1.1. ¿Qué es UNIX?

Es sin lugar a dudas uno de los sistemas operativos más extensos y potentes que hay. Sus principales virtudes son:

**Multiusuario** En una máquina UNIX pueden estar trabajando simultáneamente muchos usuarios. Cada usuario tendrá la impresión de que el sistema es suyo en exclusiva.

**Multiproceso** Cada usuario puede estar ejecutando simultáneamente muchos procesos. Cada proceso se ejecuta independientemente de los demás como si fuese el único en el sistema. Todos los procesos de todos los usuarios son ejecutados concurrentemente sobre la misma máquina, de forma que sus ejecuciones avanzan de forma independiente. Pero también podrán comunicarse entre sí.

**Multiplataforma** El núcleo del sistema operativo UNIX está escrito en más de un 95 % en lenguaje C. Gracias a ello ha sido portado a máquinas de todos los tamaños y arquitecturas. Los programas desarrollados para una máquina UNIX pueden ser llevados a cualquier otra fácilmente.

### ¿Cómo es UNIX?

Desde el punto de vista del usuario, hay que resaltar las siguientes características:

**Sensible al tipo de letra.** Distingue entre mayúsculas y minúsculas. No es lo mismo `unix` que `Unix` que `UNIX`.



**Ficheros de configuración textuales.** Para facilitar su edición sin necesidad de herramientas sofisticadas.

**Para usuarios NO torpes.** Otros Sistemas Operativos someten cada acción “peligrosa” (Ej. reescribir un fichero) al consentimiento del usuario.

En UNIX, por defecto, se entiende que el usuario sabe lo que quiere, y lo que el usuario pide, se hace, sea peligroso o no.

**Lo borrado es irrecuperable.** Es un ejemplo del punto anterior. Hay que tener cuidado. Como veremos más adelante existe la posibilidad de que el usuario se proteja de sí mismo en este tipo de mandatos.

## ¿Cómo es LINUX?

**POSIX.** Es una versión de UNIX que cumple el estándar *Portable Operating System Interface*.

**Libre.** Se distribuye bajo licencia GNU, lo que implica que se debe distribuir con todo su código fuente, para que si alguien lo desea, lo pueda modificar a su antojo o necesidad.

**Evoluciona.** Por ser libre, está en permanente desarrollo y mejora, por programadores voluntarios de todo el mundo.

En este capítulo se presenta el entorno UNIX desde el punto de vista de un usuario del mismo. El objetivo es que el usuario se sienta mínimamente confortable en UNIX. Veremos, en este orden:

1. Usuarios y Grupos
2. Sesión
3. Mandatos
4. Procesos
5. Árbol de Ficheros
6. Descriptores de fichero
7. Intérprete de mandatos
8. Configuración

## 1.2. Usuarios y Grupos

Para que una persona pueda utilizar un determinado sistema UNIX debe haber sido previamente dado de alta como usuario del mismo, es decir, debe tener abierta una cuenta en el sistema.

### Usuario

Todo usuario registrado en el sistema se identifica ante él con un nombre de usuario (*login name*) que es único. Cada cuenta está protegida por una contraseña (*password*) que el usuario ha de introducir para acceder a su cuenta. Internamente el sistema identifica a cada usuario con un número único denominado UID (*User Identifier*).

### Grupo

Para que la administración de los usuarios sea más cómoda estos son organizados en grupos. Al igual que los usuarios, los grupos están identificados en el sistema por un nombre y por un número (GID) únicos. Cada usuario pertenece al menos a un grupo.

### Privilegios

Las operaciones que un usuario podrá realizar en el sistema estarán delimitadas en función de su identidad, esto es por la pareja UID-GID, así como por los permisos de acceso al fichero o recurso al que desee acceder.

### Superusuario

De entre todos los usuarios, existe uno, denominado “superusuario” o *root* que es el encargado de administrar y configurar el sistema.

Es aquel que tiene como UID el número 0 (cero). El superusuario no tiene restricciones de ningún tipo. Puede hacer y deshacer lo que quiera, ver modificar o borrar a su antojo. Ser superusuario es una labor delicada y de responsabilidad.

La administración de un sistema UNIX es compleja y exige amplios conocimientos del mismo.

## 1.3. Sesión

Una *sesión* es el periodo de tiempo que un usuario está utilizando el sistema. Como sabemos, cada usuario tiene una cuenta que estará protegida por una contraseña o *password*. Para poder entrar en el sistema necesitamos un *terminal* que nos permita conectarnos a la máquina UNIX en cuestión.

A veces es tan sencillo como acceder a través del interfaz gráfico del sistema operativo si usted ya dispone de una máquina UNIX y poner en marcha un programa que haga de *terminal* (por ejemplo `gnome-terminal`, `xterm`, etc.). Si es así, acceda al sistema, ponga en marcha uno de estos programas y salte a la tarea 1.3.

En general, para poder acceder a una máquina UNIX desde otra máquina (probablemente Windows), necesitará un programa de SSH (por ejemplo `Putty`, `openssh`, etc.) al


que le pedirá que se conecte a la máquina en cuestión. Deberá entonces seguir los siguientes pasos para entrar y salir del sistema:

T 1.1 `login: myname_`

*Para entrar al sistema introduzca su nombre de usuario o login name.*

T 1.2 `passwd: *****_`

*Si ya ha puesto una contraseña deberá usted introducirla ahora, para que el sistema verifique que usted es quien dice ser.*

T 1.3 

*Al entrar en el sistema se arranca un programa que nos servirá de **intérprete de mandatos**, un shell en general y Bash en los sistemas más actuales (aunque es configurable). Nos presenta un mensaje de “apremio” o prompt. Algunos prompts clásicos son:*

```
$ _  
> _  
k0174@triqui3:~ $ _
```

T 1.4 `logout`

*Para terminar una sesión deberá usar este mandato o bien `exit`.*


## 1.4. Mandatos

El interprete de mandatos (podrá ver esto en más detalle en la sección 1.8) de todos los sistemas operativos (Bash, Csh, Ksh, CMD, etc.) admite mandatos que facilitan el uso del sistema. Su forma general es:


`mandato [opciones] [argumentos...]`

Los campos están separados por uno o más espacios. Algo entre corchetes [ ] es opcional. La notación ... indica uno o varios. Las opciones generalmente comenzarán por un - (menos).


A lo largo de esta presentación se irá solicitando que usted realice numerosas prácticas, observe el comportamiento del sistema o conteste preguntas...

T 1.5 

*Ahora debe usted entrar en el sistema como se indicó en el apartado anterior.*

T 1.6 

*Observe que UNIX distingue entre mayúsculas y minúsculas.*

T 1.7 

*Conteste, ¿sabría usted terminar la sesión?*

...en muchos casos se sugerirá que ejecute ciertos mandatos...



T 1.8 `who am i`  
*Obtenga información sobre usted mismo.*

T 1.9 `date`  
*Conozca la fecha y hora.*

`cal` T 1.10  
*Visualice un calendario del mes actual.*

...se darán más explicaciones para ampliar sus conocimientos...

La mayoría de los mandatos reconocen las opciones `-h` o `--help` como petición de ayuda sobre él mismo.

...y se solicitará que realice más practicas...

`date --help` T 1.11  
*Aprenda qué opciones admite este mandato.*

`?` T 1.12  
*¿Qué día de la semana fue el día en que usted nació?*

...así mismo, durante esta presentación iremos introduciendo un resumen de algunos mandatos que se consideran más importantes.

Si desea información completa sobre algo, deberá usar el mandato `man`.

`man [what]` \_\_\_\_\_ *Manual Pages*

Visualiza una copia electrónica de los manuales del sistema. Es seguro que nos asaltará a menudo la duda de como usar correctamente algún mandato o cualquier otra cosa. Deberemos consultar el manual.

Los manuales se dividen en secciones:

- 1 Mandatos (Ej. `sh`, `man`, `cat`).
- 2 Llamadas al sistema (Ej. `open`, `umask`).
- 3 Funciones de librería (Ej. `printf`, `fopen`).
- 4 Dispositivos (Ej. `null`).
- 5 Formato de ficheros (Ej. `passwd`).
- 6 Juegos.
- 7 Miscelánea.
- 8 Mandatos de Administración del Sistema.

`man -h` T 1.13  
*Busque ayuda breve sobre el mandato `man`.*

`man man`

T 1.14

*Lea la documentación completa del mandato `man`.*

`man [1-8] intro`

T 1.15

*En caso de conflicto, si ejemplo existe en diferentes secciones la misma hoja de manual, deberá explicitar la sección en la que buscar.*

*La hoja `intro` de cada sección informa de los contenidos de la misma.*

### Algunos mandatos útiles

A continuación se citan algunos de los muchísimos mandatos disponibles en UNIX. Si desea más información sobre alguno de ellos, consulte el manual.

`pwd` Visualización del directorio actual de trabajo.

`cd` Cambio del directorio actual de trabajo.

`mkdir` Creación de directorios.

`rmdir` Eliminación de directorios.

`echo` Muestra el texto que se indique por su salida.

`env` Muestra el valor de las variables de entorno.

`ls` Listado del contenido de un directorio.

`cat` Visualización del contenido de un fichero.

`pico` Un sencillito editor de ficheros.

`vi` El editor de ficheros más extendido en entorno UNIX.

`emacs` El editor de ficheros más potente disponible.

`rm` Eliminación de fichero.

`cp` Copiar ficheros.

`mv` Mover o renombrar ficheros.

`wc` Contar las palabras y líneas de un fichero.

`grep` Búsqueda de una palabra en un texto.

`sort` Ordenación de ficheros.

`find` Búsqueda de ficheros que cumplen determinadas características.

`chmod` Cambio de los permisos de un fichero.

`sleep` Simplemente espera que transcurra los segundos indicados.

`id` Muestra la identidad del usuario que lo ejecuta.

T 1.16



*¿Cuántas “palabras” contiene el fichero `/etc/password` del sistema?*

## 1.5. Procesos

Cando usted invoca un mandato, el **fichero ejecutable** del mismo nombre es ejecutado. Todo programa en ejecución es un *proceso*. Todo proceso se identifica por un número único en el sistema, denominado PID (*Process Identifier*).

### Concurrencia

En todo momento, cada proceso activo de cada usuario del sistema, esta compitiendo con los demás por ejecutar. El sistema operativo es el que marca a quién le toca el turno en cada momento.

Todo esto sucede muy muy deprisa, con el resultado de que cada proceso avanza en su ejecución sin notar la presencia de los demás, como si fuera el único proceso del sistema, pero en realidad todos están avanzando simultáneamente. A esta idea se le denomina *concurrencia*.

<code>ps</code>	Permite ver una instantánea de los procesos en el sistema. Admite multitud de opciones tal y como muestra su página de manual.	T 1.17
<code>man ps</code>		T 1.18
<code>top</code>	Permite observar la actividad de los procesos en el sistema con cierta periodicidad.	T 1.19

### Jerarquía de procesos

Todos los procesos del sistema se organizan en una jerarquía *padre-hijo(s)* que tiene como ancestro último al proceso denominado *init* cuyo PID es 1.

Todo proceso tiene también asociado un número que identifica a su proceso padre, es el PPID.

<code>pstree</code>	Permite ver una instantánea de la jerarquía de procesos en el sistema. Admite multitud de opciones.	T 1.20
---------------------	---	--------

## 1.6. Árbol de Ficheros

Existe una única estructura jerárquica de nombres de fichero, es decir, **no existe el concepto de “unidad”**. En UNIX los dispositivos se “montan”.

### Jerarquía de directorios

A continuación se presenta cual sería la jerarquía de directorios básica presente en una máquina Linux. Sólo se presentan algunos de los directorios de primer y segundo nivel.

```

/          Directorio raíz.
|-- bin    Ejecutables globales (públicos).
|-- dev    Ficheros especiales (representan dispositivos).
|-- etc     Ficheros de configuración globales.
|-- home    Contiene las cuentas de usuario.
|-- lib     Librerías básicas globales (públicas).
|-- proc    Información instantánea sobre el sistema y sus procesos.
|-- root    Cuenta del superusuario.
|-- sbin    Ejecutables para administración.
|-- tmp     Espacio para ficheros temporales (público).
|-- usr     Segundo nivel. No imprescindible para el arranque.
|   |-- bin
|   |-- doc        Documentación básica.
|   |-- etc
|   |-- include    Ficheros de cabecera (.h) del lenguaje C.
|   |-- lib
|   |-- local      Tercer nivel. Añadidos a la instalación básica.
|   |-- man        Manuales del sistema.
|   |-- sbin
|   |-- share      Ficheros compartidos.
|   '-- tmp
'-- var         Spollers, cerrojos, correo, log, etc.

```

## Directorio raíz

La raíz de la jerárquica de nombres es el denominado directorio raíz y se denota con el carácter / (dividido por) **no por el \ (*backslash*)**.

## Directorio HOME

Nada más entrar en el sistema, el usuario es situado en el denominado directorio HOME de su cuenta.

Cada cuenta tiene su propio HOME, que es creado por el administrador del sistema al abrirle la cuenta al usuario.

Este directorio pertenece al usuario correspondiente. Por debajo de él podrá crear cuantos subdirectorios o ficheros quiera.

## Tipos de objeto

Para averiguar qué es o qué hay contenido bajo un nombre de fichero puede usar el mandato `file`.

Existen (básicamente) 3 tipos de “objetos”, a saber:

**Directorios:** Es un contenedor cuyas entradas se refieren a otros ficheros y/o directorios.

El uso de los directorios da lugar a la jerarquía de nombres. Todo directorio contiene siempre las siguientes dos entradas:

- (*punto*) Se refiere al mismo directorio que la contiene.

.. (*punto punto*) Se refiere al directorio padre de este, es decir, a su padre en la jerarquía.

**Ficheros normales:** Están contenidos en los directorios. Contienen secuencias de bytes que podrían ser códigos de programas, datos, texto, un fuente en C o cualquier otra cosa.

**Ficheros especiales:** Son como ficheros normales, pero no contienen datos, sino que son el interfaz de acceso a los dispositivos periféricos tales como impresoras, discos, el ratón, etc, etc.


Cada dispositivo, existente o posible, se haya representado por un fichero especial que se encuentra bajo el directorio `/dev`.


UNIX trata estos ficheros especiales (y por lo tanto a los dispositivos) exactamente igual que trata a los ficheros normales, de forma y manera que para los programas, los unos y los otros son **indistinguibles**.

`pwd` \_\_\_\_\_ *Print Working Directory*

Permite averiguar cuál es el directorio en que nos encontramos en cada momento, al cuál denominamos directorio actual de trabajo.

`pwd` T 1.21  
*Visualice su directorio actual de trabajo.*

 *Observe que se muestra el camino absoluto (desde el directorio raíz) de su directorio HOME.* T 1.22

 *Observe que las componente de un camino se separan entre sí con el carácter / (dividido por) y no por el \ (backslash).* T 1.23

`cd [dir]` \_\_\_\_\_ *Change Directory*

Con este mandato cambiamos de un directorio a otro. Admite como argumento el nombre del directorio al que queremos cambiar.

Existen tres casos especiales que son de utilidad.

`cd`  
Invocado sin argumento, nos devuelve directamente a nuestro HOME.

`cd .`  
El directorio de nombre `.` (*punto*) hace referencia siempre al directorio actual de trabajo. Realmente no nos va a cambiar de directorio.

`pwd` T 1.24  
*Observe que permanecemos en el mismo directorio.*

Como veremos más adelante, el directorio `.` resulta útil para referirnos a ficheros que “están aquí mismo”.

`cd ..`

Todo directorio contiene siempre una entrada de nombre `..` (punto punto) que hace referencia siempre al directorio padre del directorio actual.

T 1.25

`pwd`

*Observe que ascendemos al directorio padre.*

T 1.26

`cd ..`

*Repita la operación hasta llegar al raíz.*

T 1.27

`pwd`

*Observe que no se puede ir más allá del directorio raíz.*

T 1.28

`cd`

*¿A qué directorio hemos ido?*

El directorio HOME es siempre accesible mediante el símbolo `~`.

T 1.29

`cd /etc`

*Muévase al directorio `/etc`.*

T 1.30

`pwd`

*Compruebe en qué directorio está ejecutando el intérprete.*

T 1.31

`cd ~`

*Vuelva al directorio HOME.*

T 1.32

`pwd`

*Compruebe de nuevo en qué directorio está ejecutando el intérprete.*

`ls [-opt] [dirs...] _____ List Directory Contents`

Este mandato nos presenta información sobre los ficheros y directorios contenidos en el(os) directorio(s) especificado(s).

Si no se indica ningún directorio, informa sobre el directorio actual de trabajo.

Este mandato admite cantidad de opciones que configuran la información obtenida. Entre ellas las más usadas son, por este orden:

`-l` Formato “largo”. Una línea por fichero o directorio con los campos: permisos, propietario, grupo, tamaño, fecha y nombre.

`-a` Informa también sobre los nombres que comienzan por `.` (punto), que normalmente no se muestran.

`-R` Desciende recursivamente por cada subdirectorio encontrado informando sobre sus contenidos.

Si necesita explorar más opciones, consulte el manual.

T 1.33

`ls`

*Observe que su directorio HOME parece estar vacío.*

T 1.34

`ls -a`

*Pero no lo está. Al menos siempre aparecerá las entradas `.` y `...`*

T 1.35



*Es posible que aparezcan otros nombres que con `ls` no vio.*

*Este mandato evita mostrar aquellas entradas que comienzan por el carácter punto. Podríamos decir que los ficheros o directorios cuyo nombre comienza por un punto están "ocultos", pero se pueden ver si usamos la opción `-a`.*

*Suelen ser ficheros (o directorios) de configuración.*

`ls -la`

T 1.36

*Si queremos conocer los detalles del "objeto" asociado a cada entrada de un directorio, usaremos la opción `-l`.*



*¿A quién pertenecen los ficheros que hay en su HOME?*

T 1.37

*¿Qué tamaño tienen?*

*¿Cuándo fueron modificados por última vez?*

`ls /home`

T 1.38

*Observe el HOME de otros usuarios de su sistema.*

*Inspeccione el contenido de sus directorios.*

`ls -la /`

T 1.39

*Muestra el "tronco" del árbol de nombres.*

`man 7 hier`

T 1.40

*Lea una descripción completa de la jerarquía del sistema de ficheros UNIX que está usando.*

`ls -??? /bin`

T 1.41

*Obtenga el contenido del directorio `/bin` ordenado de menor a mayor el tamaño de los ficheros.*



*Observe que aparece un fichero de nombre `ls`. Ese es el fichero que usted está ejecutando cada vez que usa el mandato `ls`.*

T 1.42

*Esos ficheros son sólo una parte de los que usted podría ejecutar, hay muchos más.*



*¿A quién pertenece el fichero `/bin/ls`?*

T 1.43



*Sus permisos permiten que usted lo ejecute.*

T 1.44

## 1.7. Descriptores de fichero

Los procesos manejan ficheros a través de los denominados descriptores de fichero. Por ejemplo, el resultado de abrir un fichero es un descriptor. Las posteriores operaciones de manejo (lectura, escritura, etc.) de este fichero reciben como parámetro el descriptor.

Desde el punto de vista de nuestro programa, un descriptor no es más que un número entero positivo (0, 1, 2, ...).

## Descriptores estándar

Los tres primeros descriptores de fichero son los denominados “estándar”, y se considera que siempre están disponibles para ser usados.

Normalmente estarán asociados al terminal pero podrían estar asociados a un fichero o a cualquier otro objeto del sistema de ficheros, pero este es un detalle que no ha de preocuparnos a la hora de programar.

Los programas que se comporten de forma “estándar” los utilizarán adecuadamente.

**Entrada estándar.** Es la entrada principal al programa. Los programas que se comportan de forma estándar reciben los datos que han de procesar leyendo de su entrada estándar (descriptor de fichero número 0).

**Salida estándar.** Es la salida principal del programa. Los programas que se comportan de forma estándar emiten sus resultados escribiendo en su salida estándar (descriptor de fichero número 1).

**Error estándar.** Es la salida para los mensajes de error del programa. Si un programa que se comporta de forma estándar ha de emitir un mensaje de error deberá hacerlo a través de su error estándar (descriptor de fichero número 2).

## 1.8. Intérprete de mandatos

La interacción del usuario con el sistema UNIX puede ser a través de un interfaz gráfico, pero lo más común es que sea a través de un interfaz textual, esto es, a través del diálogo con un intérprete de mandatos.

En UNIX se denomina *shell* (cáscara) al intérprete de mandatos, esto es, el programa capaz de interpretar y poner en ejecución los mandatos que el usuario teclea. No existe un único *shell* sino muchos.

T 1.45 `ls -l /bin/*sh`

*Los ficheros que aparecen son los diferentes shell disponibles. Quizás aparezca algún otro fichero. Quizás aparezcan enlaces simbólicos (otro tipo de objeto del sistema de ficheros).*

Se distinguen fundamentalmente en la sintaxis que reconocen y en las facilidades que proporcionan para su uso interactivo y para usarlos como lenguaje de programación de “guiones” de mandatos (*scripts*).

**sh** Es el denominado *Bourne Shell*. Es el primer *shell* que se desarrolló para UNIX, y es el preferido para la programación de *scripts*.

**csh** El “C” *shell*. Se llama así porque la sintaxis que reconoce se aproxima a la del lenguaje C. También es uno de los primeros *shell* desarrollados para UNIX, y se ha preferido para uso interactivo.

**bash** *Bourne Again Shell*, es una versión más evolucionada del **sh**, más apta para el uso interactivo.



**tcsh** Versión mejorada del **csch**, con más facilidades para el uso interactivo como completar nombres de fichero y edición de línea de mandatos.

otros Existen otros muchos: **ksh**, **zsh**, **rc**, etc. (para gustos, colores).

Ya hemos estado usando el *shell*, de hecho, observe que si está usted en su cuenta, entonces **está usted ejecutando uno de ellos en este momento**, probablemente Bash.



## 1.9. Variables de entorno

Además de los argumentos que indicamos en su invocación, cada mandato que ejecutemos recibirá una serie de parejas "nombre=valor", que son las denominadas variables de entorno.

Las variables de entorno son pasadas de proceso padre a proceso hijo por un mecanismo de herencia.

Las variables del entorno se utilizan para configurar ciertos comportamientos de los procesos. Por poner algunos ejemplos citaremos las siguientes:

**HOME** Su valor es el nombre del directorio raíz de nuestra cuenta en el sistema.

**PATH** Enumera un conjunto de directorios donde se localizará los ejecutables que invoquemos.

**TERM** Indica el nombre o tipo del terminal que estemos usando.

**USER** Indica el nombre del usuario que somos.

**SHELL** Es el nombre completo del intérprete de mandatos que estemos usando.

Existen otras muchas más variables de entorno, y podríamos situar en el entorno cualquier otra información que sea de nuestro interés.

Podremos visualizar las variables de entorno existentes haciendo uso del mandato **env**.

**env**

T 1.46

*Visualice todas las variables de entorno.*

Para visualizar el valor de una variable de entorno determinada se puede usar el mandato **echo** junto con la expansión de dicha variable.

**echo Hola mundo**

T 1.47

*Logre que el intérprete de mandatos imprima por pantalla **Hola mundo**.*

**echo \$HOME**

T 1.48

*Imprima el valor de la variable **HOME**.*



*¿Qué shell está usted usando?*

T 1.49

## Variable de entorno PATH

Para entender el comportamiento del *shell* es conveniente profundizar en la misión de la variable de entorno PATH.


Cuando le pedimos al *shell* que ejecute un mandato, este lo buscará en los directorios indicados en la variable de entorno PATH y sólo en ellos. Es conveniente que la variable PATH se refiera al menos a los directorios más comunes, esto es, `/bin`, `/usr/bin` y `/usr/local/bin`.

**Por razones de seguridad, es altamente recomendable que el directorio actual (“.”) NO esté incluido en la variable PATH.**

T 1.50  ¿Por qué?


## 1.10. Ficheros de texto ejecutables

Si un fichero de texto tiene permisos de ejecución (`man chmod`) y su primera línea reza `#! ejecutable`, podremos ejecutar directamente este fichero, y su contenido será leído e interpretado por el ejecutable. De esta manera podemos crear nuestros propios ficheros de mandatos ejecutables, que denominamos “guiones” (*scripts*).

T 1.51  Escriba un fichero cuya primera línea sea `#! /bin/cat`, y luego añada las líneas de texto de desee. Dele permisos de ejecución y ejecútelo.

Como podrá observar, un fichero de texto ejecutable no tiene porqué estar necesariamente asociado a un *shell*, sino que puede asociarse a cualquier mandato que haga uso de su entrada estándar.

Además en la primera línea podemos añadir opciones del mandato (una opción como máximo, aunque puede ser compuesta).

T 1.52  Modifique el fichero anterior para que al ser ejecutado cuente las líneas y las palabras de sí mismo, pero **no** los caracteres. Para ello consulte el manual sobre el mandato `wc`.

Este mecanismo, de ficheros de texto ejecutables nos permitirá la realización de ficheros de mandatos, que serán interpretados por el *shell* de nuestra elección.

## Capítulo 2

# Uso del *Shell*

En este apartado exploraremos el uso interactivo del intérprete de mandatos. Vamos a ver algunas características básicas del *shell* que nos serán de mucha utilidad.

### 2.1. El *prompt*

Cuando el usuario entra en su cuenta a través de un terminal, el sistema arranca un nuevo intérprete de mandatos asociado a dicho terminal. Un *shell* arrancado de esta manera está en modo interactivo y nos informará de ello mostrando en el terminal un *prompt* o mensaje de apremio. Un *shell* está a nuestro servicio.

En este documento se muestran los diálogos con el *shell* de la siguiente manera:

```
$ _
```

Se utilizará habitualmente será el símbolo \$ como *prompt* del shell.

### 2.2. Estado de terminación

Todo mandato UNIX termina devolviendo un valor que permite identificar su estado de terminación. Un valor distinto de 0 indica, por convención, que ocurrió un error de algún tipo, mientras que un valor 0 indica que funcionó como se esperaba.

Para conocer el estado de terminación del último mandato utilizado se puede usar la variable de entorno “?”.

```
ls
```

T 2.1

*Ejecute un mandato que no falle.*

```
echo $?
```

T 2.2

*Compruebe el estado de terminación de dicho mandato, debería ser 0.*

```
ls Supercalifragilisticexpialidocious
```

T 2.3

*Ejecute un mandato que falle.*

```
echo $?
```

T 2.4


*Compruebe el estado de terminación de dicho mandato, debería ser distinto de 0.*

Los programadores deben ser consistentes con esta convención por lo que tendrán que asegurarse de devolver un estado de terminación significativo cuando escriben programas.

## 2.3. Primer y segundo plano

Hasta ahora hemos venido arrancando mandatos que “toman el control del terminal” hasta que terminan. Se dice de ellos que ejecutan en primer plano (*foreground*). Podemos solicitar la ejecución de varios mandatos en primer plano invocando cada uno en una línea diferente, pero también podríamos hacerlo separándolos con el carácter ‘;’.

T 2.5 `sleep 3; ls /; sleep 3; who`

T 2.6  *Observe como se ejecuta un mandato después del anterior en estricta secuencia. Observe cómo se respetan los tiempos entre mandatos.*

En UNIX es posible solicitar la ejecución de mandatos en segundo plano (*background*) poniendo el carácter & al final de los mismos. El mandato se ejecutará en concurrencia con otros mandatos que ejecutemos en primer o segundo plano a continuación.

T 2.7 `sleep 3 & ls / & sleep 3 & who &`

T 2.8  *Observe como se ejecutan concurrentemente y la salida de los mandatos se mezcla.*

## 2.4. Redirección

En UNIX podemos redirigir la entrada estándar, salida estándar y/o error estándar de los mandatos. Esta es una característica muy muy útil, ya que, sin necesidad de modificar en absoluto un mandato podemos usarlo para que tome su entrada de un fichero en vez de tener que teclearla (el teclado es habitualmente la entrada estándar de los programas que ejecutemos desde el intérprete).

De igual manera en vez de que la salida del mandato aparezca en la pantalla (salida estándar), podríamos enviarla a un fichero para usarla con posterioridad.

La sintaxis correcta para realizar redirecciones es añadir al final del mandato la redirección con la siguiente notación:

`mandato < fichero`

En su ejecución, el mandato usará **fichero** como entrada, leerá de él y no del teclado.

`mandato > fichero`

En su ejecución, la salida del mandato será enviada a **fichero** y no a la pantalla. Si **fichero** no existiera con anterioridad, será creado automáticamente, pero si existe, será previamente truncado a tamaño cero.

`mandato >> fichero`

Añade la salida del mandato al final del fichero indicado.

`mandato 2> fichero`

Los mensajes de error que un mandato pueda generar también pueden ser redirigidos a un fichero, pero normalmente preferimos que se visualicen por la pantalla.

`mandato 2>&1`

Redirige la salida de error a donde quiera que esté dirigida la salida estándar.



Obtenga en un fichero de nombre `todos_los_ficheros` un listado en formato largo de todo el árbol de ficheros recorriéndolo recursivamente desde el directorio raíz.

T 2.9

Dado que este mandato puede llevar un rato en ejecutar, lo arrancamos en segundo plano, para poder seguir realizando otros mandatos.

`ls -l todos_los_ficheros`

T 2.10

Compruebe que tamaño del fichero está aumentando.



Aparecerán en pantalla sólo los mensajes de error que el mandato produce al no poder acceder a ciertos directorios.

T 2.11

`grep login_name < todos_los_ficheros > mis_ficheros`

T 2.12

Use el mandato `grep` para filtrar las líneas del fichero de entrada que contienen el texto indicado, en este caso su nombre de usuario. Deje el resultado en un fichero de nombre `mis_ficheros`.

`ls Supercalifragilisticexpialidocious > ficheroraro`

T 2.13



¿Qué ha ocurrido? ¿Cuál es el contenido del fichero `ficheroraro`?

T 2.14

`ls Supercalifragilisticexpialidocious 2> ficheroraro`

T 2.15



¿Qué ha ocurrido esta vez? ¿Cuál es el contenido del fichero `ficheroraro`? Explíquelo.

T 2.16

## 2.5. Secuencia de mandatos

### *Pipeline*

En UNIX podemos redirigir la salida estándar de un mandato directamente a la entrada estándar de otro. Con ello construimos secuencias de mandatos que procesan el resultado del anterior y emiten su resultado para que sea procesado por el siguiente mandato de la secuencia.


Esto lo conseguimos separando los mandatos con el carácter “|” (*pipe*).


Esta es una facilidad muy poderosa de UNIX, ya que, sin necesidad de ficheros intermedios de gran tamaño podemos procesar grandes cantidades de información.

```
ls -la /home | grep login_name
```


T 2.17

*Obtiene un listado en formato largo de todos los directorios de cuentas y filtra aquellas líneas que contienen nuestro nombre de usuario.*

T 2.18  *No se crea ningún fichero intermedio.*

T 2.19  *Obtenga el listado “todos sus ficheros” de la sección anterior, conectando el resultado al mandato **grep** para quedarse sólo con sus ficheros. Mande el resultado a un nuevo fichero de nombre **MIS\_ficheros**. Póngalo así, con mayúsculas, para no machacar el fichero nombrado así pero en minúsculas.*

*Recuerde invocar el mandato con **&** para poder seguir haciendo otras cosas.*

T 2.20  *Cuando hayan terminado de ejecutar los mandatos, compruebe las diferencias entre los ficheros de nombre **mis\_ficheros** y **MIS\_ficheros**. Para ello utilice el mandato **diff**, consultando el manual si es necesario.*

## Secuencias condicionales

Es también posible separar mandatos o secuencias con los siguientes símbolos:

**&&** Solicitamos que el mandato o secuencia a la derecha del símbolo sólo se ejecute si el de la izquierda terminó correctamente.

**||** Solicitamos que el mandato o secuencia a la derecha del símbolo sólo se ejecute si el de la izquierda terminó incorrectamente.

T 2.21 

```
true && echo "El anterior terminó bien"
```

T 2.22 

```
false || echo "El anterior terminó mal"
```

## 2.6. Metacaracteres

Los metacaracteres (*wildcards*) son caracteres comodín. Con su uso podemos hacer referencia a conjuntos de ficheros y/o directorios, indicando la expresión regular que casa con sus nombres.

**?** Comodín de carácter individual.

**\*** Comodín de tira de cero o más caracteres.

**~** Abreviatura del directorio *home*.

**~user** Abreviatura del directorio *home* de *user*.

**[abc]** Casa con un carácter del conjunto especificado entre los corchetes.

[*x-y*] Casa con un carácter en el rango especificado entre los corchetes.

[*^...*] Niega la selección indicada entre los corchetes.

{*str,...*} Agrupación. Casa sucesivamente cada tira.



*Obtenga un listado en formato largo, de los ficheros de los directorios /bin y /usr/bin/ que contengan en su nombre el texto sh.* T 2.23










*Obtenga un listado en formato largo, de los ficheros del directorio /dev que comiencen por tty y no terminen en dígito.* T 2.24

## 2.7. Uso interactivo

Un buen *shell* para uso interactivo ofrece servicios que le facilitan la tarea al usuario. Esto significa que existen múltiples combinaciones de teclas asociadas con alguna funcionalidad especial. Para conocerlas completamente deberá consultar el manual.



### Edición de línea

Podemos mover el cursor () sobre la línea de mandatos que estamos tecleando con  , borrar antes y sobre el cursor con  y , ir al inicio y al final con  y , etc.

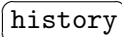


*Observe que pasa si intenta editar una línea más larga que la pantalla.* T 2.25

### Histórico de mandatos

Si hace poco que tecleamos un mandato no es preciso volverlo a teclear. Basta con buscarlo haciendo uso de las flechas del teclado  y .

El mandato interno **history** nos informa de todos los mandatos que hemos usado últimamente.



*Visualice los mandatos que hasta ahora ha realizado en su shell.* T 2.26



*Observe que **history** guarda los mandatos anteriores de una sesión otra.* T 2.27



*Repita el último mandato.* T 2.28



*Repita el mandato que ocupa la posición 5 en la historia.* T 2.29

## Fin de datos

La combinación de teclas `Ctrl D` en una línea vacía indica “fin de datos”.

Si la pulsamos y hay algún programa leyendo del terminal, entenderá que no hay más datos y terminará normalmente.


T 2.30 `cat`

*Es un mandato que lo que lee de su entrada lo escribe en su salida. Escriba varias líneas de datos para observar su comportamiento.*

*Como está leyendo del terminal, para que termine normalmente, deberá indicarle “fin de datos”.*

T 2.31 `$ Ctrl D`

*El shell también es un programa que lee de su entrada estandar. Indíquele “fin de datos”.*

T 2.32 

*El shell entenderá que no hay más datos, terminará normalmente y por lo tanto nos sacará de la cuenta.*

*Si ha sido así y este comportamiento no le agrada, hay una forma de configurar el shell para evitar que suceda. Más adelante la veremos.*

## Control de trabajos

En UNIX podemos arrancar mandatos para que se ejecuten en segundo plano (*background*) poniendo el carácter `&` al final de la línea.

También, desde su *shell* puede suspender el trabajo que está en primer plano (*foreground*), rearrancar trabajos suspendidos, dejarlos en segundo plano, matarlos, etc.

`Ctrl C` Pulsando esta combinación de teclas cuando tenemos un programa corriendo en primer plano, le mandaremos una señal que “lo matará”.


`Ctrl Z` El programa en primer plano quedará suspendido y volveremos a ver el *prompt* del *shell*.

`jobs` Este mandato interno nos informa de los trabajos en curso y de su estado. Podremos referirnos a uno concreto por la posición que ocupa en esta lista.

`fg` Ejecutándolo, el último proceso suspendido volverá a primer plano. Para referirnos al segundo trabajo de la lista de trabajos usaremos `fg %2`.

`bg` Ejecutándolo, el último proceso suspendido volverá a ejecutar, pero en segundo plano! Es como si lo hubiéramos arrancado con un `&` detrás.

`kill` Este mandato interno permite enviarle una señal a un proceso, con lo que, normalmente, lo mataremos.

T 2.33 

*Arranque varios procesos en background, por ejemplo varios `sleep 60 &`, y a continuación consulte los trabajos activos.*





Arranque un proceso en primer plano, por ejemplo `cat` y suspéndalo. Consulte los trabajos activos. Vuelva a ponerlo en primer plano y luego mándele una señal que lo mate.

T 2.34



¿Cómo lo mataría si se encuentra en segundo plano?

T 2.35

## Completar nombres de fichero

Si tenemos un nombre de fichero escrito “a medias” y pulsamos el tabulador (`↩`), el `tcsh` intentará completar dicho nombre. Si lo que pulsamos es la combinación `Ctrl D`, nos mostrará la lista de nombres que empiezan como él.

También funciona para nombres de mandato.

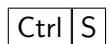


Escriba `ls .cs↩` y luego intente `ls .cs Ctrl D`, observe la diferencia.

T 2.36

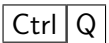
## Control de *scroll* de terminal

Denominamos terminal al conjunto pantalla/teclado o a la ventana a través de la cual dialogamos con el *shell*.



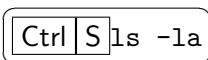
Congela el terminal. No mostrará nada de lo que nuestros programas escriban, ni atenderá a lo que tecleemos.

Si por descuido pulsamos `Ctrl S` podríamos pensar que el sistema se ha quedado colgado, pero no es así.



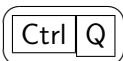
Descongela el terminal. Lo que los programas hubiesen escrito mientras la pantalla estuvo congelada, así como lo que hubiésemos tecleado, aparecerá en este momento.

El uso común de la pareja `Ctrl S` `Ctrl Q` es para controlar la salida de programas que emiten muchas líneas.



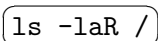
Observe que, de momento, nada de lo que hemos tecleado aparece en el terminal.

T 2.37



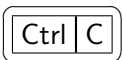
Verá como el mandato que tecleamos aparece y se ejecuta con normalidad.

T 2.38



El resultado de este mandato son demasiadas líneas de texto. Juegue con `Ctrl S` `Ctrl Q` a parar y dejar correr la pantalla.

T 2.39



Cuando se canse, mate el trabajo.

T 2.40

## 2.8. Algunos mandatos útiles

`less [files...]` \_\_\_\_\_ *On Screen File Browser*

Filtro para la visualización por pantalla de ficheros. Es una versión mejorada del clásico mandato `more`.

Si lo usamos indicando un conjunto de ficheros, podremos pasar al siguiente y al anterior con `:n` y `:p` respectivamente.

Con este mandato sólo podremos ver los ficheros, pero no modificarlos, más adelante veremos algún editor.

T 2.41 `less *_ficheros`

*Visualice el contenido de los ficheros que existen ya en su HOME que terminan por “\_ficheros”.*

T 2.42 `ls *`

*Observe que el metacaracter `*` no alude a los ficheros cuyo nombre empieza por punto.*

T 2.43 `ls -aLR | less`

*Liste todos los ficheros del sistema pero visualícelos por páginas.*

`passwd` \_\_\_\_\_ *Change User Password*

UNIX protege el acceso a las cuentas de los usuarios mediante una palabra clave.

Las cuentas del sistema, sus contraseñas y resto de información, están registradas en el fichero `/etc/passwd`.

T 2.44 `less /etc/passwd`

*Visualice el contenido del fichero de contraseñas.*

*Este fichero contiene la información sobre las cuentas de usuario que hay en su sistema.*

*Compruebe que existe una cuenta con su nombre de usuario.*

T 2.45 `man 5 passwd`

*Estudie cuál es el contenido del fichero de contraseñas.*

Un usuario puede cambiar su palabra clave haciendo uso del mandato `passwd`. Es muy importante que la contraseña no sea trivial, ni una fecha, ni una matrícula de coche, teléfono, nombre propio, etc.

T 2.46 `passwd`

*¡Si no tiene un buen password, cámbielo ahora mismo!*

`cp orig dest` \_\_\_\_\_ *Copy Files*

Obtiene una copia de un fichero o bien copia un conjunto de ficheros al directorio indicado.

Admite variedad de opciones, que no vamos a describir en este documento (si tiene dudas, consulte el manual).

Interesa destacar que este **es un mandato peligroso**, dado que (por defecto) hace lo que le pedimos sin consultar, lo cuál puede suponer que machaquemos ficheros que ya existan. Para que nos consulte deberemos usarlo con la opción `-i`.

```
cp /etc/passwd ~
```

T 2.47

*Copie en su HOME el fichero /etc/passwd.*



*Visualice que su contenido es realmente una copia del original.*

T 2.48



*Copie el fichero .cshrc sobre el que ahora tiene en su cuenta con el nombre passwd.*

T 2.49



*¿Le ha consultado cp antes de reescribir? ¿Se ha modificado el fichero?*

T 2.50



*Intente las mismas operaciones usando la opción -i.*

T 2.51

`rm files...` \_\_\_\_\_ *Remove Files*

Borrar ficheros. Para ver sus opciones consulte el manual.

También **es un mandato peligroso**. En este caso, `rm` no sólo hace lo que le pedimos sin consultar, sino que en UNIX **no existe manera de recuperar** un fichero (o directorio) que haya sido borrado. Por lo tanto se hace imprescindible usarlo con la opción `-i`.



*Intente borrar el fichero passwd de su cuenta con `rm -i passwd`. Conteste que no.*

T 2.52



*Borre el fichero passwd que ahora tiene en su cuenta.*

T 2.53



*Observe que efectivamente ha sido eliminado.*

T 2.54



*Intente borrar el fichero /etc/passwd.*

T 2.55



*¿Puede hacerlo? ¿Porqué?*

T 2.56

`mv orig dest` \_\_\_\_\_ *Move (Rename) Files*

Cambia el nombre de un fichero o bien mueve un fichero o conjunto de ficheros a un directorio.

También **es un mandato peligroso**, porque podría “machacar” el(os) fichero de destino sin pedirnos confirmación. Se debe usar con la opción `-i`.



*Intente mover el fichero /etc/passwd a su HOME.*

T 2.57



Intente mover algún fichero de su cuenta (uno que no le valga) al directorio `/etc`. T 2.58



T 2.59 ¿Puede hacer estas operaciones? ¿Porqué?



T 2.60 Cambie de nombre al fichero `mis_ficheros` a `MIS_ficheros`.



T 2.61 ¿Pudo hacerlo? ¿Le consultó?

`mkdir dir _____` *Make Directory*

Crea un nuevo directorio. Por supuesto, si ya existe un fichero o directorio con dicho nombre dará un error.

Los directorios que usted cree serán suyos. Sólo podrá crear ficheros o directorios en aquellos directorios donde usted tenga permiso de escritura.



T 2.62 Cree en su cuenta un subdirectorio de nombre `subarbol`.



T 2.63 Visualice su contenido con `ls -la`.



T 2.64 Cree otros subdirectorios más profundos, pero sin hacer uso del `cd`.

T 2.65 `ls -laR ~`

Visualice el contenido actual de su HOME recursivamente.

`rmdir dir _____` *Remove Directory*

Borra un directorio. Por supuesto el directorio debe estar vacío.



T 2.66 Intente borrar un directorio **no** vacío, Ej. `subarbol`.



T 2.67 ¿Puede hacerlo?



T 2.68 Visualice el contenido de `subarbol`.



T 2.69 Elimine los directorios más profundos, y vaya ascendiendo, hasta conseguir eliminar el directorio `subarbol`.

## 2.9. Configuración

Cada usuario puede configurar el comportamiento de su cuenta editando convenientemente ciertos ficheros presentes en su HOME que dependen de qué *shell* use usted.

El fichero de configuración que es de nuestro interés en este momento es:

`.bashrc` Se ejecuta siempre que el usuario arranca un nuevo `bash`.

`.cshrc` Se ejecuta siempre que el usuario arranca un nuevo `csh` o `tcsh`.

Dado que estos ficheros sólo se leen al arrancar el *shell*, para que los cambios que hagamos en ellos tengan efecto, deberemos bien salir de la cuenta y volver a entrar o bien hacer uso del mandato interno `source` para que su *shell* lea e interprete el contenido del fichero.

### Mandatos peligrosos

En UNIX, si se borra un fichero o un directorio no hay forma de recuperarlo. Además, los mandatos obedecen ciegamente a quien los usa, y no consultan. Para evitar meter la pata es conveniente usar ciertos mandatos de forma que consulten antes de actuar. Para ello, vamos a usar el mandato interno `alias`, para protegernos de los mandatos problemáticos.

### Fin de datos

Evitaremos que el *shell* termine a la pulsación `Ctrl` `D` pidiéndole que lo ignore.

#### `alias`

T 2.70

*Ejecutando este mandato observará que alias están en funcionamiento.*

*Los alias modifican “al vuelo” los mandatos que queremos ejecutar, justo antes de ser finalmente ejecutados.*

#### `pico file`

T 2.71

*Es un editor de uso muy sencillo. Úselo para realizar la siguiente modificación en el fichero de configuración de su shell.*

*Escoja entre la versión de la izquierda, si usted usa el `tcsh` o el `csh`, o la de la derecha si usa el `bash`. Añada al final del fichero de configuración indicado las siguientes líneas:*

`.cshrc`

```
alias cp 'cp -i'
alias mv 'mv -i'
alias rm 'rm -i'
set ignoreeof
```

`.bashrc`

```
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
IGNOREEOF=10
```

#### `source`

T 2.72

*Recuerde que debe pedirle a su shell que lea e interprete el contenido del fichero de configuración, para que tenga constancia de la modificación que acaba de hacer.*

`alias`

T 2.73

*Observará si los alias que ha establecido están en funcionamiento.*

T 2.74

`Ctrl` `D`

*Compruebe si esta secuencia hace terminar su shell.*

## Capítulo 3

# Programación de *Scripts*

En este apartado usaremos el *Bash* para programar ficheros de mandatos (*scripts*), pero en ocasiones, para probar un mandato complejo o un *script* sencillo, lo usaremos de forma interactiva.

Simplemente ejecutaremos el mandato `sh` y este nos devolverá el carácter `$` que es *prompt* característico del `sh`. Para salir de nuevo al `cs`h pulsaremos *Ctrl-D* o bien usaremos el mandato interno `exit`.

### Ejemplo

```
user@hostname $
user@hostname $ ps
  PID TTY STAT  TIME COMMAND
 1154 p1 S    0:02 -csh
 2556 p1 R    0:00 ps
user@hostname $ echo Mi PID es $$
Mi PID es 1154
user@hostname $ date
Thu Jul 30 18:20:37 CEST 1998
user@hostname> who am i
hostname.domain!user  ttyt1    Jul 30 11:23 (:0.0)
user@hostname $ sh
$ ps
  PID TTY STAT  TIME COMMAND
 1154 p1 S    0:02 -csh
 2561 p1 S    0:00 sh
 2562 p1 R    0:00 ps
$ echo $$
2561
$ ^D
user@hostname $
```

### 3.1. Edición

Para poder realizar los ejercicios de este capítulo será necesario utilizar un editor de texto plano. Le recomendamos que utilice uno de los editores clásicos de UNIX (**vi**, **emacs**, **pico**, **nano**, etc.). El uso de todos los editores mencionados se basan en la activación de comandos (borrar, avanzar línea, seleccionar un texto, etc.) mediante la combinación de teclas. Usted tendrá que adaptarse al uso de este tipo de editores antes de continuar con los ejercicios.

### 3.2. Comentarios

Tanto el programa en su conjunto como cada una de sus partes o funciones deben estar bien documentados, a base de comentarios que clarifiquen qué hace y cómo lo hace.

Los comentarios son cualquier texto, comenzando con el carácter almohadilla (**#**) y terminando al final de la línea. `Esto se ejecuta. # Esto no.`

### 3.3. Variables y Entorno


El *shell* proporciona variables cuyo valor son tiras de caracteres. El nombre de una variable podrá contener letras, dígitos y el carácter '\_', pero no puede empezar por '\_'. Las minúsculas y las mayúsculas se consideran distintas.


Para dar valor a una variable escribiremos `variable=valor`, sin espacios en blanco! Para obtener el valor de una variable usaremos la notación `$variable` o `${variable}` si a continuación del nombre de la variable viene letra o dígito.

El *entorno* es un subconjunto de las variables del *shell* que son heredadas por todo proceso derivado de este. Una variable puede ser añadida al conjunto de variables de entorno con el mandato `export variable`. Las variables de entorno pueden ser visualizadas con el mandato `env`.

T 3.1  Escriba el mandato identidad con el siguiente contenido y ejecútelo:

```
#!/bin/bash
# identidad
# Identifica al usuario usando las variables de entorno.
echo Usted es el usuario \"$USER\".
```

T 3.2  Como podrá observar el Bash lee, interpreta y ejecuta el contenido del fichero. Este pequeño programa hace uso además de la variable de entorno `USER`.

T 3.3  Amplíe el programa anterior para que visualice una descripción completa del usuario y la máquina basándose en la información contenida en el entorno.



### 3.3.1. Variable PATH

Cuando le pedimos al *shell* que ejecute un mandato, lo intentará encontrar entre sus mandatos internos y entre las funciones que tengamos definidas. Si no se encuentra, lo buscará en el conjunto de directorios indicados en la variable de entorno PATH y sólo en ellos.

Dado que un *script* puede fallarle a alguna persona si tiene un PATH diferente o incompleto, se recomienda incluir en una correcta especificación de esta variable de entorno.

```
PATH=/usr/bin:/bin
```



*Experimente con el siguiente mandato en PATH:*

T 3.4

```
#!/bin/bash -x
# en_PATH
# Experimento sobre la variable PATH.
# El valor de esta variable afecta a la
# localización de los mandatos que usamos.
PATH=""; ls # No se encuentra.
PATH=""; /bin/ls # SI se encuentra.
PATH=/bin; ls # SI se encuentra.
```



*Observe que la opción -x del shell permite “depurar” nuestro programa.*

T 3.5

### 3.3.2. Variables especiales

La denominadas variables especiales, no pertenecen al entorno. Es el *shell* quien les da valor.

\$0 Nombre del *script*.

\$9 Parámetro del *script* o función en la posición indicada (1 a 9).

\$# Número de parámetros posicionales o argumentos.

\$\* Lista de argumentos.

\$@ Lista de argumentos. Pero "\$@" no es una tira de caracteres, sino copia exacta de la lista de argumentos

\$\$ Identificador del propio proceso *shell*.

\$? Valor devuelto por el último mandato ejecutado.

#! Identificador del último proceso lanzado en segundo plano.



*Codifique el siguiente mandato argumentos que visualiza el valor de todas las variables especiales relativas a los argumentos. Experimente la diferencia entre "\$@", "\$@", "\$\*" y \$\*.*

T 3.6

```

#!/bin/bash
# argumentos args...
# Experimento sobre variables especiales relativas a argumentos.

echo '"$@"' vale "$@", y si lo recorro obtengo:
for ARG in "$@"; do echo __${ARG}__; done

echo '$@' vale $@, y si lo recorro obtengo:
for ARG in $@; do echo __${ARG}__; done

echo '"$*"' vale "$*", y si lo recorro obtengo:
for ARG in "$*"; do echo __${ARG}__; done


echo '$*' vale $*, y si lo recorro obtengo:
for ARG in $*; do echo __${ARG}__; done

```

*Ejecútelo con los siguientes argumentos.*

```
argumentos 1 "Dos 2" "3 III" 'Cuatro ****'
```

T 3.7  ¿Dónde y cuándo se expanden los comodines de 'Cuatro \*\*\*\*'?

T 3.8  ¿Porqué es precisa la notación \${ARG}?

T 3.9 

*El mandato interno `shift` desplaza los parámetros posicionales un lugar a la izquierda, perdiéndose el antiguo valor de \$1, que pasará a valer lo que valía \$2, etc. y decrementa \$#.*

*Añada al final del mandato `argumentos` un último recorrido del siguiente modo:*

```

echo Y si voy desplazando los parámetros con "shift" obtengo:
while [ $# -ne 0 ]      # Equivalente a...      until test $# -eq 0
do
    echo __${1}__
    shift
done

```

### 3.4. El mandato test

Es uno de los mandatos más usados en *shell scripts*, estúdielo en profundidad puede serle muy útil `man test`. Puede ser un mandato interno del *shell* o ser un mandato externo.

```
ls -l /usr/bin/[ /usr/bin/test
```

Puede usarse de dos maneras: `test expression` o `[ expression ]`. Observe que sus argumentos (así como el carácter `]`) deben estar convenientemente separados por espacios.

Un buen *shell script* debe verificar que los argumentos con los que ha sido invocado (si es preciso alguno) son correctos, y si no lo son debe indicarle al usuario la forma correcta de utilización. Para verificar el número de argumentos usaremos el mandato `test`.



*Amplíe el siguiente mandato `test_test` para que realice todos los posibles `test` de uno o dos argumentos.* T 3.10

```
#!/bin/bash
# test_test arg1 [arg2]
# Usa el mandato test(1) de todas las formas posibles
# sobre 1 o 2 argumentos.

test_as_file()
{
    for opt in b c d e f g
    do
        test -$opt "$1" && : TRUE || : false
    done
}

if [ $# -eq 2 ]
then
    set -x          # Activamos el modo depuración.
    test_as_file "$1"
    test_as_file "$2"
    ...
elif [ $# -eq 1 ]
then
    ...
else
    ...
fi
```

Ejécútelo con... `"", /dev/console, ., 2 02, "1 -lt 3"`, etc.




*¿\$1 hace referencia al mismo parámetro cuando se usa en el programa principal que cuando se usa dentro de una función?* T 3.11



*¿Porqué es precisa la notación entrecomillada "\$1"?* T 3.12



*¿Que significa la notación `mandato && mandato || mandato`?* T 3.13

T 3.14  : es un mandato interno. ¿Para qué sirve? (`man bash` y busque :).

T 3.15  `set` es un mandato interno. ¿Para qué sirve?

### 3.5. Funciones

Usando el *Bourne Shell* podemos hacer uso de grupos de mandatos, agrupados entre llaves '`{}`' allí donde podríamos haber usado un mandato simple.


Las funciones en *Bourne Shell* son simplemente el resultado de ponerle un nombre único a un conjunto de mandatos agrupados mediante llaves. Son una de las características más poderosas del *sh* y es necesario acostumbrarse a usarlas.

```
function() {  
    # Conjunto de mandatos  
    # que implementan la función  
    return 0 # terminación correcta  
}
```

- En una función los parámetros posicionales `$1`, `$2`, etc. son los argumentos de la función (no los argumentos del *script*).
- Una función puede ser usada de igual modo que se puede usar un mandato externo.
- Una función devolverá su propio estado de terminación usando la sentencia `return`. Si se usa `exit` dentro de una función, se estará terminando el *script* completo, no solo la función.

```
return 0    # Terminación correcta
```

- Las funciones son ideales para encapsular una determinada labor. Se pueden redirigir su entrada, salida o error, conectarlas por pipes, etc.
- Las funciones del *Bourne Shell* **pueden ser recursivas!**

T 3.16  Escriba el mandato árbol que reciba como argumentos una lista de ficheros y/o directorios e imprima los nombres completos de cada fichero y recorra cada directorio de forma recursiva en profundidad.

- El resultado debe ser un nombre completo de fichero por línea, convenientemente endentado (con tabuladores) para indicar la profundidad de la recursión.
- Utilice una función recursiva de nombre `sub_arbol` que reciba como primer argumento una tira de tabuladores, como segundo el nombre del directorio que se está explorando en este momento y a continuación cualquier número de nombres de fichero o directorio a evaluar.
- Para obtener el contenido de un directorio utilice '`ls -A1 $DIR`'.

### 3.6. Entrada, salida y error estándar

La entrada, salida y error estándar son los descriptores de fichero 0, 1 y 2. Cada uno tiene una misión concreta y deben ser usados convenientemente.

Error Los mensajes de error deben aparecer en el estándar error y no en la salida estándar.

```
echo Fichero de configuración inexistente. >&2
```

A veces, por estética, interesará silenciar posibles mensajes de error generados por los mandatos que usamos. Lo conseguiremos dirigiendo la salida de error de dicho mandato a `/dev/null`.

```
expr $NUM 2>/dev/null || reintentar
```

Salida La salida estándar es para los mensajes que van dirigidos al usuario o a otros procesos.

```
echo -n "Desea crearlo?[s] "
```

Entrada La entrada estándar es para recoger la respuesta de los usuarios o la salida de otros procesos.

```
read RESPUESTA
```

```
CWD='pwd'
```

Interactivo Es posible que la entrada estándar no esté asociada al terminal (haya sido redirigida) y no haya un usuario al cual consultar. Si es preciso nuestros programas pueden tener esto en cuenta y comportarse de forma distinta según estén siendo usados de forma interactiva o no `tty -s`.

*in-situ* Podemos incorporar en nuestro propio *script* el texto que queremos que sea la entrada a un mandato.

```
cat <<'Fin_ayuda'
```

Las líneas de texto siguientes, hasta una que literalmente rece `Fin_ayuda` serán la entrada al mandato. Si la marca de “fin de documento *in-situ*” presenta algún *quoting* el texto será literal, si no serán expandidas las variables y los mandatos entre comillas inversas.



*Escriba el mandato **reto** que retará al usuario a que adivine un número (por ejemplo el PID del proceso). Para ello leerá los valores numéricos que el usuario teclee y responderá indicando si el número secreto es mayor, menor o igual, en cuyo caso terminará.*

T 3.17

- Para comparar los valores numéricos haga uso del mandato `test`.
- El programa sólo debe admitir ser usado de forma interactiva.
- El programa debe llamar al usuario por su nombre haciendo uso de la variable `USER`.
- Debe evitar morir si el usuario le manda una señal desde teclado. Para ello capturará las señales `SIGINT` y `SIGQUIT` (`man 7 signal`) haciendo uso del mandato interno `trap`.
- Si el usuario responde al reto con `Ctrl-D` (fin de datos), el programa pedirá una confirmación positiva antes de terminar.

### 3.7. Manipulación de tiras de caracteres

Internamente el *shell* ofrece pocos mecanismos para manipular tiras de caracteres, si bien siempre se puede hacer uso de mandatos externos capaces de manipular ficheros de texto de forma sofisticada.

**case** La sentencia **case** permite localizar patrones sencillos en tiras de caracteres. Se usa por ejemplo para analizar los parámetros de entrada de los *shell scripts* y reconocer opciones.

```
case $1 in
  0) echo Cero ;;
  1|2|3|4) echo Positivo ;;
  -[1234]) echo Negativo ;;
  *.* ) echo Decimal ;;
  *) echo Otra cosa ;;
esac
```

**read** El mandato interno **read** lee una línea de su entrada estándar y asigna valor a las variables que se le pasan como argumentos. La línea se divide por los caracteres que se consideran separadores de campo que normalmente son el tabulador y el espacio.

**set** El mandato interno **set**. Si le pasamos una lista de argumentos, los separa por los caracteres que se consideran separadores de campo y los asigna como valores a los parámetros posicionales **\$1**, **\$2**, etc. y actualiza **\$#**.

Además, el mandato **set** sirve para activar opciones del propio *shell*, como el modo depuración, modo *verbose*, etc. Consulte el manual.

**IFS** El conjunto de caracteres que el *shell* considera separadores de campo son los que contiene la variable de entorno **IFS** (*Input Field Separator*). Cambiando el valor de esta variable podemos dividir líneas por los caracteres que queramos.

```
IFS=: read field_1 field_2 rest <file
```

```
IFS=', ' set "1,2,3,4 5.6"
```

Observe en el ejemplo anterior la forma de asignar valor a una variable en la misma línea en la que se invoca al mandato, sin separarlos por **;**. Usando esta sintaxis la modificación del valor de la variable sólo afecta a dicha invocación, y no es permanente.

T 3.18



Documente adecuadamente el siguiente código. Para ello, consulte el manual hasta entender cómo se consigue el valor adecuado en cada caso.

```
#!/bin/bash -x
# hora
# Experimento manipulación de tiras de caracteres.
TIME='date | cut -c12-19'
TIME='date | cut -d\ -f4'
```

```

TIME='date | sed 's/. * . * \(. *\ ) . * . */\1/'
TIME='date | awk '{print $4}''
TIME='set `date`; echo $4'
TIME='date | (read u v w x y z; echo $x)'
TIME='date +%T'

```



Escriba el mandato **adivina** que tendrá que intentar adivinar el valor de un parámetro numérico que sólo el usuario conoce, preguntándole cosas que sólo debe responder con si o no.

T 3.19

- El programa sólo debe admitir ser usado de forma interactiva.
- El programa debe presentar un menú de los conceptos que sabe adivinar (edad, altura, etc.) y leer la opción que el usuario seleccione. Para identificar la opción utilice la sentencia **case**.
- Escriba la función auxiliar *pregunta\_si* que reciba como argumentos el texto de una pregunta, se lo presente al usuario, lea su respuesta (usando el mandato interno **read**) y con un **case** compruebe si la respuesta es afirmativa. Si lo es terminará bien, sino terminará mal.
- Escriba la función auxiliar *dicotomía* que controle dos valores numéricos menor y mayor que representan un rango, y pregunte al usuario sobre el valor medio del rango, de forma que se pueda reducir el rango a la mitad. Para calcular el valor medio utilice el mandato externo **expr**.



Codifique el programa **ckpw** que realice ciertas verificaciones de seguridad relativas al fichero **/etc/passwd**.

T 3.20

- Debe leer cada línea del fichero **/etc/passwd** separándola en sus componentes (login, password, uid, gid, name, home y shell).
- Comprobará que el password no está vacío.
- Que el uid y el gid son numéricos.
- Que el home es un directorio que no tiene permisos de escritura para el usuario del programa.
- Que el shell existe como fichero ejecutable y que está catalogado como shell válido en el fichero **/etc/shells**.