1. Primeros pasos en Unix

1.1. Entrada en el sistema

Para empezar a trabajar en una máquina Unix, el primer paso que tenemos que realizar es *conectarnos* a ella y *autentificarnos* como usuarios válidos.

A la hora de conectarnos tenemos básicamente dos opciones:

- Si el sistema Unix está instalado en una máquina *local* y trabajamos desde la consola o desde una terminal local, bastará con encender el terminal y ya podremos autentificarnos, dado que la máquina nos pedirá el nombre de usuario y la contraseña.
- Si accedemos a una máquina remota, deberemos utilizar algún programa de comunicaciones (*telnet*, *ssh*, *putty*, ...) para llevar a cabo la conexión. Una vez ejecutado el programa e indicado la dirección o nombre de la máquina a la que queremos conectarnos, el programa nos ofrecerá una ventana similar a la que nos encontraríamos tras encender el terminal en la opción anterior.

Una vez que ya estamos conectados a una máquina Unix, bien desde una terminal o desde una máquina remota, ya podemos empezar a trabajar. En nuestro caso, nos conectaremos a una máquina Linux de nombre *petra.* Tras conectarnos, se nos mostrará la pantalla de bienvenida del sistema, en la que se nos está pidiendo un nombre de *login*, donde tenemos que introducir el nombre de nuestra cuenta:

```
login as:
```

Al ser el sistema operativo Unix un sistema multiusuario, sólo los usuarios autorizados tienen acceso a la máquina. Para identificar a cada usuario, cada uno tiene un nombre de cuenta (*login name*), de tal manera que cada usuario tendrá acceso solamente a aquellos recursos a las que esté autorizado.

Una vez introducido el nombre de la cuenta, el sistema solicitará una contraseña (*password*), palabra que sólo debe conocer el propietario de la cuenta, dado que se utiliza para autentificar la identidad del usuario que va a entrar en sesión.

Si en alguno de los dos pasos anteriores (si se ha introducido un nombre de cuenta inexistente o se ha tecleado mal la contraseña), el sistema impedirá la entrada a ese usuario, quedando constancia en el sistema de ese hipotético intento de intrusión en el sistema (la mayoría de los casos va a ser debido a confusiones al teclear, pero también puede ser alguien intentando entrar sin estar autorizado).

Si todo ha transcurrido bien, tras introducir la contraseña el sistema mostrará una serie de mensajes, tras lo cual aparecerá un signo de dólar ($), que es el *prompt* del sistema, que indica que el sistema se encuentra dispuesto a recibir las ordenes del usuario[1].

Por razones de seguridad, es aconsejable cambiar la contraseña regularmente (o al menos la primera vez que se entra), para evitar que posibles intrusos puedan hacerse con la palabra de paso de algún usuario y una vez dentro del sistema causen algún problema, que sería achacado al propietario de la cuenta. Por ello, es aconsejable utilizar una contraseña que sea difícil de averiguar.

La orden para cambiar de contraseña es la orden `passwd`:

```
petra:~$ passwd
Changing password for albizu
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
petra:~$
```

Como se puede observar, en primer lugar, se pide la contraseña actual. Esto se hace así para asegurarse que el que está cambiando la password es el propietario de la cuenta, y no un extraño aprovechando un descuido del propietario que ha abandonado su terminal estando en sesión. En ningún caso se muestra por pantalla ninguna de las contraseñas que se introducen, como medida de precaución para evitar posibles "mirones".

Una vez verificado este punto se pide que se introduzca la contraseña nueva, tras lo cual se pide que se repita, para evitar errores en el tecleo. Si se repite mal la contraseña, ésta permanece inalterada.

Por motivos de seguridad, es conveniente seguir una serie de normas a la hora de elegir una contraseña, para que ésta sea difícil de averiguar:

- Debe tener un tamaño mínimo determinado (en nuestro sistema Linux debe tener como mínimo 6 caracteres y como máximo 8).
- Debe evitarse usar únicamente de letras, y en el caso que se utilicen sólo letras, deben mezclarse mayúsculas y minúsculas.
- No deben utilizarse palabras que guarden relación con el nombre de la cuenta ni con ningún dato de su propietario (nombre, apellidos, teléfono, matrícula del coche, ...), dado que son datos que pueden ser averiguados por hipotéticos intrusos. De hecho, hay programas caza-contraseñas que se basan en la información del usuario almacenada en el sistema para intentar averiguar contraseñas de usuarios.

1.2. La orden man

Uno de las ordenes más útiles del Unix puede que sea `man` (de manual). Dado que hay multitud de órdenes en el sistema, que suelen tener unos nombres más bien crípticos, y que además suelen contar con multitud de opciones, sólo los usuarios que llevan mucho tiempo trabajando con el sistema pueden recordar los nombres y las opciones más usuales de las ordenes que habitualmente utilizan.

---

[1] Esto ocurrirá cuando entremos en un sistema con intefaz de usuario textual. Si el sistema al que accedemos dispone de un interfaz gráfico de usuario nos aparecerá el escritorio del mismo, con los distintos menús y elementos que lo compongan. Dado que estudiaremos las órdenes del shell textual de Unix, será preciso en este caso ejecutar la aplicación “terminal” para disponer de un intérprete de órdenes textual.

Dado que Unix se diseñó desde el principio como un sistema operativo donde los usuarios se iban a conectar remotamente, sus diseñadores pensaron que el típico manual impreso que acompañaba a los sistemas en aquella época no iba a ser de utilidad (se necesitaría una copia para cada usuario). En su lugar, el sistema se acompaña, como ayuda a los usuarios, de un completo manual en línea de todas las ordenes del sistema, de tal manera que en cualquier momento puede un usuario consultar el formato completo y la funcionalidad de cualquier orden. Hay que hacer notar que el sistema de manual de un sistema Unix incluye información no sólo de las órdenes de usuario, sino también de las llamadas al sistema, de estructuras de datos que se utilizan, etc. Además, cualquier aplicación que se instale suele venir acompañada de sus páginas de manual, con lo que a través de una única orden puede buscarse ayuda de casi todo lo relativo al sistema.

Para consultar el manual del sistema basta con introducir la orden `man orden`, que da como resultado la salida por pantalla de las páginas del manual disponibles para la `orden` introducida.

La estructura típica de la ayuda acerca de una orden es la siguiente:

```
RMDIR(1)                        User Commands
RMDIR(1)

NAME
       rmdir - remove empty directories

SYNOPSIS
       rmdir [OPTION]... DIRECTORY...

DESCRIPTION
       Remove the DIRECTORY(ies), if they are empty.

       --ignore-fail-on-non-empty

              ignore each failure  that  is  solely  because  a directory is
              non-empty

       -p, --parents

              Remove DIRECTORY and its ancestors. E.g., `rmdir -p a/b/c'  is
              similar to `rmdir a/b/c a/b a'.

       -v, --verbose

              output a diagnostic for every directory processed

 Manual page rmdir(1) line 1
```

Donde habitualmente suelen aparecer algunas de las siguientes secciones:

- **NAME**: Indica el nombre de la orden, y una pequeña descripción de la funcionalidad de la misma. En ocasiones aparece el nombre de más de una orden, de tal manera que se agrupan ordenes estrechamente relacionadas dentro de la misma página del manual.
- **SYNOPSIS**: Presenta el formato completo de la orden, indicando de manera resumida todas las opciones que tiene. Las partes encerradas entre corchetes indican que esa parte es opcional.
- **DESCRIPTION**: Comenta en detalle la funcionalidad completa de la orden.
- **OPTIONS**: Indica detalladamente el funcionamiento de cada una de las opciones que tiene la orden.

- **NOTES**: Presenta información adicional que no se ha considerado oportuna incluirla dentro de la descripción de la orden.
- **SEE ALSO**: Expone una serie de ordenes relacionadas, que puede que sea interesante consultar.

Opcionalmente pueden aparecer otras partes de la página, como **WARNINGS**, **BUGS**, **FILES**, **AUTHORS**, ..., donde aparecerá información acerca de avisos a tener en cuenta a la hora de ejecutar la orden, errores conocidos, ficheros relacionados con la orden, los autores de la orden, etc.

Para movernos a lo largo de la información que nos presenta la orden `man`, que puede ocupar varias páginas, disponemos de varias opciones:

- Flecha arriba: desplaza el texto una línea hacia arriba.
- Flecha abajo: desplaza el texto una línea hacia abajo.
- Espacio o Control-f : desplaza el texto una página hacia abajo.
- Control-b : desplaza el texto una página hacia arriba.
- /*texto*: Busca el texto indicado dentro de la información mostrada por la orden.
- *n*: Repite la última búsqueda realizada, desde el punto en que nos encontramos y hacia delante.
- *b*: Repite la última búsqueda realizada, desde el punto en que nos encontramos y hacia atrás.
- *q*: Sale de la orden `man`, devolviéndonos al prompt del sistema.

A pesar de ser una orden muy potente y útil, tiene un pequeño inconveniente de funcionamiento, que puede hacernos desesperar cuando más necesitados estamos de ayuda. Cuando no sabemos exactamente de qué orden queremos ayuda, `man` no nos sirve para nada:

```
petra:~$ man password
No existe entrada de manual para password
petra:~$
```

De todas maneras, tenemos otra opción. Existe otra orden (`apropos`), que nos busca todas las ordenes relacionadas con algún concepto determinado. Así, si queremos saber las ordenes que hay relacionadas con “password”, por ejemplo, podemos introducir la orden

```
petra:~$ apropos password
chage (1)            - change user password expiry information
chpasswd (8)         - update password file in batch
crypt (3)            - password and data encryption
d_passwd (5)         - The dialup shell password file
getpwuid (3)         - get password file entry
grpconv (8)          - convert to and from shadow passwords and groups.
grpunconv (8)        - convert to and from shadow passwords and groups.
passwd (1)           - change user password
passwd (5)           - The password file
....

petra:~$
```

De esta manera, podemos recordar cual era la orden de la cual queríamos ayuda, pudiendo recurrir a `man` posteriormente para ampliar la ayuda para esa orden. En realidad, lo que hace la orden `apropos` es buscar la palabra que introducimos como parámetro

("password" en el ejemplo anterior) dentro del campo `name` que hemos visto antes, mostrando aquellas entradas donde aparezca.

Existe otra orden de comportamiento parecido `apropos`, como es la orden `whatis`. En este caso la orden sólo busca dentro del nombre de la página del `man`, a diferencia del caso anterior donde se buscaba dentro del nombre y la breve descripción en cada página del `man`. De entre las opciones que tiene esta orden podemos destacar la opción *w*, que permite utilizar caracteres comodines como parte del nombre de la información a buscar:

```
albizu@petra:~$ whatis man
man (1)              - an interface to the on-line reference manuals
man (7)              - macros to format man pages
albizu@petra:~$ whatis -w  man*
man (1)              - an interface to the on-line reference manuals
man (7)              - macros to format man pages
man2html (8)         - UNIX Man Page to HTML translator
mandb (8)            - create or update the manual page index caches
manpath (1)          - determine search path for manual pages
manpath (5)          - format of the /etc/manpath.config file
```

Estos apuntes no pretenden ser una guía completa acerca de las ordenes de Unix, de tal manera que nos limitaremos a señalar para cada orden que se estudie su filosofía de funcionamiento y las opciones más usuales y útiles, pudiendo hacer uso de la orden man para conocer en la profundidad que en cada caso se necesite cada una de las órdenes.

1.3. La orden info

A pesar de que a través de la orden *man* tenemos acceso a todo el manual del Unix, en algunas ocasiones la información que obtenemos no es demasiado legible. Si bien es cierto que las páginas del man están estructuradas en una serie de partes que permiten, en la mayoría de los casos, obtener información de una manera rápida y fiable, en ocasiones ocurre que la información referida a un concepto determinado es muy grande. Por ejemplo, si buscamos ayuda sobre el programa `bash` (que es el shell o intérprete de órdenes que más se utiliza hoy en día en los sistemas Linux), nos podemos encontrar con lo siguiente:

```
    ...

       Commands  inside  of  $(...)  command substitution are not
       parsed until substitution is attempted.  This  will  delay
       error  reporting  until  some  time  after  the command is
       entered.

       Array variables may not (yet) be exported.

GNU Bash-2.05a           2001 November 13                BASH(1)
 Manual page bash(1) line 5127/5151 (END)
```

Tras haber introducido la orden `man bash` nos hemos situado en la última línea de la página. Como podemos ver, esta página en concreto tiene 5151 líneas (alrededor de 75 páginas) con información sobre la orden. Este es un problema importante que tiene la orden `man`: el que la información sobre un concepto determinado esté almacenada en una única página. La lectura de esa ingente cantidad de información resulta muy ingrata, con lo que sería deseable disponer de alguna otra forma de visualizarla.

Para eso surge la orden `info`. De manera similar al `man`, `info` nos facilita ayuda sobre las ordenes del sistema, pero lo hace de una manera mucho más amigable, sobre todo para el caso donde la información a mostrar sea mucha. En el caso de la orden `info`, la información sobre cada concepto u orden está estructurada en forma de grafo, donde cada nodo es una página que además de información contiene una serie de punteros o enlaces

a otras páginas, pudiendo "navegar" por toda la ayuda de una manera más intuitiva. Si buscamos ayuda con esta orden sobre el *bash*, lo que obtendremos será lo siguiente:

```
File: bashref.info,  Node: Top,  Next: Introduction,  Prev: (dir),  Up: (dir)

Bash Features
*************

   This text is a brief description of the features that are present in
the Bash shell.

   This is Edition 2.5a, last updated 13 November 2001, of `The GNU
Bash Reference Manual', for `Bash', Version 2.05a.

   Copyright (C) 1991, 1993, 1996 Free Software Foundation, Inc.

   Bash contains features that appear in other popular shells, and some
features that only appear in Bash.  Some of the shells that Bash has
borrowed concepts from are the Bourne Shell (`sh'), the Korn Shell
(`ksh'), and the C-shell (`csh' and its successor, `tcsh'). The
following menu breaks the features up into categories based upon which
one of these other shells inspired the feature.

   This manual is meant as a brief introduction to features found in
Bash.  The Bash manual page should be used as the definitive reference
--zz-Info: (bash.info.gz)Top, 68 lines --Top-----------------------------------
-
Welcome to Info version 4.1. Type C-h for help, m for menu item.
```

En la primera línea de la pantalla vemos información que aparece en todas las pantallas de `info`: fichero de ayuda que se está utilizando (en este caso *bashref.info*), lugar donde estamos actualmente (nodo *top*, o raíz), y adonde iremos en caso de elegir la opción siguiente (*Next*), anterior (*Prev*) o hacia arriba (*Up*). Pulsando la inicial correspondiente (n, p, u) se mostrará la información del nodo siguiente (podemos realizar un recorrido secuencial de la información), anterior (nodo del que venimos) o el que está arriba en el árbol en el que está estructurada la información. La palabra *(dir)* en algún enlace representa al directorio, o lista de órdenes de las cuales se dispone información en formato `info`.

Además de poder movernos con las opciones antes vistas, con info podemos acceder directamente a las distintas secciones en que está estructurada la información utilizando los menús que aparecen en cada nodo. En el ejemplo anterior, si nos movemos con las flechas de movimiento del cursor un poco hacia abajo nos encontraremos con la siguiente pantalla:

```
* Menu:

* Introduction::                An introduction to the shell.

* Definitions::                 Some definitions used in the rest of this
                                manual.

* Basic Shell Features::        The shell "building blocks".

* Shell Builtin Commands::      Commands that are a part of the shell.
...
```

Colocando el cursor debajo de uno de los elementos del menú, y pulsando intro accederemos directamente a esa sección de la información. Por ejemplo, si nos situamos sobre *Definitions* y pulsamos intro pasaremos directamente a la pantalla:

```
File: bashref.info,  Node: Definitions,  Next: Basic Shell Features,  Prev:
Introduction,  Up: Top

Definitions
***********

   These definitions are used throughout the remainder of this manual.

`POSIX'
     A family of open system standards based on Unix.  Bash is
     concerned with POSIX 1003.2, the Shell and Tools Standard.

`blank'
     A space or tab character.

...
```

De esta manera, podemos acceder de una forma mucho más rápida y eficiente a la parte de la ayuda que nos interesa ver en cada momento. Véase cómo en esta pantalla (como en todas las de `info`) está presente la primera línea antes descrita, como lo que, por ejemplo, volver a la página anterior simplemente utilizando la opción *Up*

1.4. La orden `help`

Antes de continuar hay que estudiar un poco el funcionamiento del shell del sistema. El shell es un programa que se ejecuta en modo usuario, y que es conceptualmente muy sencillo. Es un bucle que lo único que hace es mostrar el *prompt* del sistema, recoger las órdenes del usuario, interpretarlas y ejecutarlas.

Pues bien, el shell no sabe interpretar todas las órdenes que vamos a ver aquí, ni está pensado para ser capaz de hacer un montón de cosas. Lo que sí sabe hacer el *shell* es, cuando se introduce una orden y ésta no está dentro del repertorio de órdenes que él entiende, es buscar un programa ejecutable en determinados directorios del sistema con el mismo nombre que la orden que ha introducido el usuario. Si lo encuentra, lo ejecuta. Si no lo encuentra, da un error.

Las órdenes que el shell sabe interpretar (porque las acciones a realizar ante esas órdenes están programadas dentro del propio *shell*) se denominan ***órdenes internas*** (en inglés, *built-in commands).* Las que son la ejecución de programas externos al shell se denominan, por este motivo, ***órdenes externas***.

La orden `man` vista anteriormente sirve para obtener información sobre órdenes externas y, en general, sobre cualquier programa instalado en el sistema, así como para conocer el formato de llamadas al sistema, formatos de ficheros utilizados, etc. Pero para lo que no sirve es para obtener información de órdenes internas de una manera sencilla. Si intentamos hacerlo, nos encontramos con la siguiente respuesta (la orden `cd` es una orden interna):

```
petra:~$ man cd
No existe entrada de manual para  cd
petra:~$
```

No existe una orden externa denomina cd, por lo cual no hay una página del `man` para ella.

Si queremos obtener información sobre las órdenes internas existe la orden `help`, que, por cierto, es una orden interna del shell:

```
petra:~$ help cd
cd: cd [-PL] [dir]
    Change the current directory to DIR.  The variable $HOME is the
    default DIR.  The variable CDPATH defines the search path for
```

```
    the directory containing DIR.  Alternative directory names in CDPATH
    are separated by a colon (:).  A null directory name is the same as
    the current directory, i.e. `.'.  If DIR begins with a slash (/),
    then CDPATH is not used.  If the directory is not found, and the
    shell option `cdable_vars' is set, then try the word as a variable
    name.  If that variable has a value, then cd to the value of that
    variable.  The -P option says to use the physical directory structure
    instead of following symbolic links; the -L option forces symbolic links
    to be followed.
petra:~$
```

Teniendo todo esto en cuenta, cuando busquemos información de una orden y no sepamos si es interna o externa, lo más aconsejable es probar tanto con `help` como con `man` para ver si esa orden existe, bien dentro del shell, bien como un programa del sistema independiente.

Existe una manera alternativa (y más sencilla) para conocer si una orden es interna o externa, y es utilizar la orden interna `type`. Esta orden muestra cuál va a ser el comportamiento del shell cuando introduzcamos una orden. Así, si introducimos el nombre de una orden externa tras el `type` éste contestará:

```
albizu@petra:~$ type man
man is hashed (/usr/bin/man)
 Lo que viene a decir que cuando se introduzca la orden man se ejecutará el
fichero /usr/bin/man
```

Si hacemos lo mismo con la orden *help* obtendremos:

```
albizu@petra:~$ type help
help is a shell builtin
```

Lo que nos confirma que `help` es una orden interna (*a shell builtin*).

1.5. Salida de sesión

De la misma manera que es necesario "presentarse" al sistema antes de empezar a trabajar en él, también es preciso "despedirse" de él una vez que hemos terminado nuestra sesión de trabajo. Esto es especialmente importante teniendo en cuenta que, de no hacerlo, cualquiera que pase por el puesto donde hemos estado trabajando podría acceder a nuestros datos, o simplemente trabajar suplantando nuestra personalidad.

Para terminar una sesión basta con introducir la orden *logout* o bien la orden *exit*. En la mayoría de los sistemas Unix se puede abreviar esa orden introduciendo *Control-d*.

Una vez hecho esto, si estamos trabajando en una terminal aparecerá de nuevo la pantalla de presentación del sistema, mientras que si estamos accediendo desde una máquina remota la conexión termina, pudiendo desaparecer también la ventana que se había creado para emular la terminal.

1.6. Facilidades adicionales del bash

Historial

Una primera utilidad que el *bash* ofrece para la introducción rápida de ordenes es el *historial* de órdenes introducidas. Esto es especialmente útil cuando tecleamos órdenes largas y queremos volver a ejecutarlas.

Cuando introducimos una orden por teclado además de ejecutarse el sistema la registra en un fichero. El usuario puede consultar en cualquier momento las órdenes introducidas en esa sesión utilizando la orden interna `history`:

```
albizu@petra:~$ history
    1  passwd
    2  man passwd
    3  apropos passwd
    4  apropos copy
    5  info bash
    6  history
```

Esta facilidad del shell no se limita a recordar las órdenes introducidas, sino que nos permite volver a ejecutarlas sin necesidad de volver a introducirlas. Así, si quisiéramos volver a introducir una orden determinada bastaría con teclear el número de la orden anteponiéndole el signo de exclamación (!) sin ningún espacio intermedio:

```
albizu@petra:~$ !6
    1  passwd
    2  man passwd
    3  apropos passwd
    4  apropos copy
    5  info bash
    6  history
    7  history
```

Una forma alternativa de utilizar el historial de introducción de órdenes es utilizando las flechas de movimiento del curso hacia arriba y hacia abajo. Esto nos permite "navegar" por el historial de órdenes, mostrándonos en cada caso el shell en la línea de órdenes la correspondiente orden del historial (si pulsamos una vez ↑ nos mostrará la última orden introducida, pulsándola otra vez la penúltima, etc). En el momento en que encontramos la orden que buscamos, basta con pulsar el Intro para volver a ejecutarla. También podemos editar la orden para corregir errores, añadir algún parámetro adicional, ... Esta facilidad es muy útil, por ejemplo, para corregir errores en órdenes largas, cuando estamos utilizando reiteradamente un conjunto de órdenes, etc.

Autocompleción

Otra facilidad que integra el *bash* es la de completar los nombres de órdenes y de ficheros durante la introducción de una orden tras el prompt. Cuando estamos introduciendo una orden y no hemos completado la escritura de la orden podemos pulsar el tabulador. Si lo que llevamos escrito es el principio solamente de una orden que el shell pueda ejecutar éste completará automáticamente el resto de la orden. Si lo que llevamos escrito cuadra con más de una orden no hace nada, pero si insistimos pulsando otra vez el tabulador nos muestra la lista de todas las órdenes que cuadran con lo que llevamos escrito.

Por ejemplo, si escribo *pa* y a continuación pulso el tabulador, parece que no pasa nada (el shell no sabe todavía que orden quiero introducir porque hay muchas que comienzan por *pa*). Si vuelvo a pulsar el tabulador, aparecen las órdenes que el shell conoce y que comienzan por *pa*:

```
petra:~$ pa <TAB> <TAB>
packinit        patbase         patclean        patlog
pager           patch           patcol          patmake
panache         patch-metamail  patdiff         patname
paperconf       patchls         patftp          patnotify
passwd          patchls-5.004   patgen          patpost
paste           patchls-5.005   pathchk         patsend
pat             patcil          patindex        patsnap
petra:~$ pa
```

Si continuamos escribiendo la orden, e introducimos una *s*, con la esperanza de que ya conozca la orden a la que nos referimos, todavía hay varias opciones

```
petra:~$ pas <TAB> <TAB>
```

```
passwd  paste
petra:~$ pas
```

Si, finalmente, pulsamos otra s y pulsamos el tabulador, ya puede completar la orden:

```
petra:~$ passwd
```

Esto es también válido para nombres de ficheros. Si el sistema espera que introduzcamos un nombre de fichero y ponemos sólo las primeras letras, pulsando el tabulador el sistema completará el nombre, en las mismas condiciones que se ha explicado anteriormente para las órdenes[2].

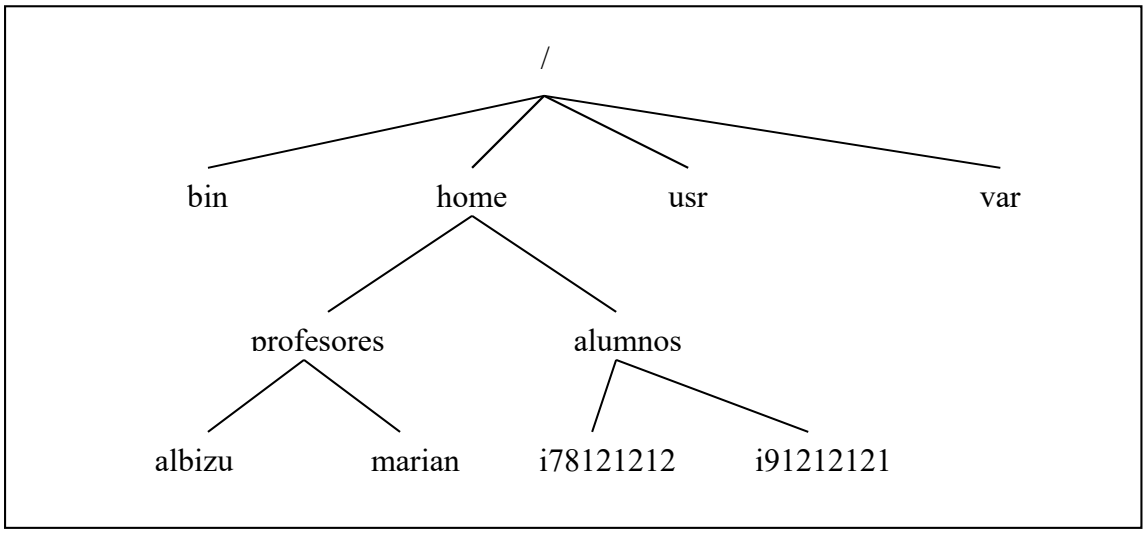
[2] Esta facilidad de autocompleción se puede configurar para que, en función de la orden, utilice una función de autocompleción diferente. Para ello se puede usar la orden interna `complete`, pero requiere privilegios admistrativos para ser usada y no es fácil de utilizar.

Ejercicios

1. Entra en petra o en el sistema Unix/Linux del que dispongas.
2. Cambia tu palabra de paso. Intenta distintas posibilidades para ver el nivel de seguridad que incorpora el sistema a la hora de permitir palabras más o menos sencillas (palabras muy cortas, sólo con letras, palabras relacionadas con el usuario, etc.)
3. Obtén ayuda de las siguientes órdenes: `man`, `passwd`, `apropos`, `help, info`, `type`.
4. ¿Qué opción del `man` hace que funcione de manera similar a la orden `apropos`?
5. ¿Para qué sirve la orden `whatis`?
6. ¿Qué fichero se ejecuta cuando introducimos la orden `man`?
7. ¿Con qué orden obtengo información sobre las órdenes internas del shell? ¿Cómo se utiliza?
8. ¿Con qué orden obtengo información sobre órdenes externas? ¿Cómo se utiliza?
9. ¿Con qué orden puedo saber si una orden es interna o externa del shell? ¿Cómo se utiliza?
10. ¿Cómo puedo saber que ordenes he introducido en el bash últimamente?
11. ¿Cómo puedo ejecutar una orden que aparece en el historial?
12. Quiero visualizar un fichero que se llama Nombredeficheromuylargodenarices. Sólo tengo en el directorio actual ese fichero que comience por N. Indique cómo puedo introducir el nombre de fichero con solo dos pulsaciones de teclado.
13. Repite la última orden introducida, utilizando las facilidades del shell. Obtén ayuda de la orden `history` (¿es interna o externa? Averígualo).
14. Utilizando las facilidades del shell, averigua qué órdenes externas comienzan con la letra p. ¿Cuántas por pa? ¿Y por pas?
15. ¿Cómo puedo repetir la ultima orden que he introducido, si estoy utilizando el bash como shell?
16. Consulta la página del man de la orden *history* para estudiar las distintas posibilidades que ofrece y averigua cómo puedes:
    a. Ejecutar la última orden introducida (sin usar la ↑).
    b. Ejecutar la orden que introdujiste hace 6 órdenes.
    c. Ejecutar la última orden que introdujiste que comenzaba por “hi”.

2. El sistema de ficheros

El sistema operativo Unix tiene un sistema de ficheros con estructura jerárquica. Esto supone que los ficheros se agrupan en directorios, pudiendo haber en un directorio otros directorios, además de ficheros. En definitiva, lo que se obtiene es una estructura en forma de árbol invertido.



Toda esta estructura parte de un directorio especial, que se denomina, por razones obvias, directorio raíz (*root*). Además, cada sesión está trabajando en un directorio de trabajo (*working directory*), o directorio actual. Además, en cada directorio hay siempre dos directorios especiales: el directorio **.**, que sirve para identificar el directorio actual; y el directorio **..**, que identifica al directorio padre del directorio actual.

Para identificar a un determinado fichero o directorio se puede utilizar su nombre completo o absoluto, que es el nombre de todos los directorios que desde el directorio raíz nos lleva a él (por ejemplo, el fichero /etc/passwd). Al igual que en el MS-DOS, en el caso de que el nombre del fichero no comience por el símbolo del directorio raíz (/) el nombre del fichero se refiere al camino desde el directorio actual.

En el ejemplo anterior, si el directorio actual es /home/profesores, para hacer referencia al directorio albizu, podemos utilizar el nombre *albizu* (como nombre relativo) o */home/profesores/albizu* (como nombre absoluto)

Nótese cómo, a diferencia del MS-DOS o del intérprete textual de los sistemas Windows, la barra que separa a los distintos directorios (e identifica al directorio raíz) es la barra de división (/), y no la utilizada en MS-DOS, que es la barra invertida (\).

Esta estructura es familiar para los usuarios del MS-DOS, que es similar, pero con una importante diferencia: en Unix existe sólo un árbol, donde se encuentran todos los discos que pudiera tener el ordenador, mientras que en MS-DOS es preciso indicar la unidad donde queremos trabajar.

Otra diferencia con el sistema operativo MS-DOS es el de las restricciones de acceso a ficheros. Dado que en un sistema Unix pueden trabajar distintos usuarios, es

recomendable proveer un mecanismo efectivo que permita distintos niveles de protección a los ficheros que haya en el disco, de tal manera que no todos los usuarios puedan realizar cualquier tipo de operación con cualquiera de los ficheros que hay en el sistema.

Las órdenes que vamos a ir introduciendo se podrán llevar a cabo siempre y cuando el usuario tenga los permisos adecuados, como estudiaremos posteriormente.

En este apartado estudiaremos en profundidad las órdenes más significativos relativos al sistema de ficheros, así como los conocimientos necesarios para poder sacarles el partido adecuado.

2.1. Órdenes de información sobre discos

Como ya hemos indicado, el sistema de ficheros de Unix tiene una única estructura jerárquica, donde radican todos los ficheros a los que tiene acceso el sistema. En el caso de que haya más de un disco en el ordenador, es necesario *montar* todos los discos sobre el sistema de ficheros principal. Esta operación de montaje (que normalmente se lleva a cabo en el arranque del sistema, o en su caso sólo la realizan usuarios autorizados), consiste simplemente en hacer corresponder un directorio en el sistema de ficheros principal al directorio raíz del disco a montar. Los usuarios normales lo único que pueden hacer al respecto es conocer los discos que están montados y donde, por medio de la orden `mount`:

```
petra:~$ mount
/dev/sda1 on / type ext2 (rw,errors=remount-ro,errors=remount-ro)
proc on /proc type proc (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda2 on /usr type ext2 (rw)
/dev/sdb1 on /var type ext2 (rw)
/dev/sdc1 on /home type ext2 (rw,nosuid,usrquota)
/dev/sdd1 on /mnt/extra type ext2 (rw,nosuid)
petra:~$
```

Otra orden que nos informa de la situación de los discos que se encuentran montados, en qué directorios están montados (*puntos de montaje*) y además nos dice el tamaño y la ocupación de cada uno de ellos es la orden `df` (*disk free*, por disco libre, dado que nos informa también de la capacidad libre de cada disco):

```
petra:~$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/sda1               253807     11475    229226   5% /
/dev/sda2             16115711    785578  14491273   5% /usr
/dev/sdb1             17072850   3072560  13111152  19% /var
/dev/sdc1             17072850  11649664   4534048  72% /home
/dev/sdd1             17072850   2696320  13487392  17% /mnt/extra
petra:~$
```

Como se puede ver en la pantalla anterior, es habitual que todos los usuarios compartan el espacio en un disco. Si por alguna circunstancia (intencionada o accidental) un usuario acapara mucho espacio en disco esto va en detrimento del espacio disponible para el resto de los usuarios. Para evitar que se dé esta circunstancia se puede instalar en el sistema un sistema de *cuotas de disco*. La cuota de cada usuario define la cantidad máxima de información que puede almacenar en un determinado disco. Para saber la cuota que tiene asignado un usuario se puede utilizar la orden `quota`:

```
petra:~$ quota
Disk quotas for user albizu (uid 1004): none
petra:~$ quota -v
Disk quotas for user albizu (uid 1004):
     Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
      /dev/sdc1  190305       0       0            1308       0       0
petra:~$
```

En primer lugar, ejecutando la orden sin opciones, muestra las cuotas especificadas para el usuario. En el caso de ejemplo, el usuario no tiene asignada cuota. Con la opción `-v` muestra más información, indicando los límites tanto de espacio en disco como de número de ficheros máximos que puede tener el usuario.

Otra orden nos permite conocer el espacio que ocupa un determinado directorio. La orden *du (*de *disk usage)* nos muestra el número de bloques (en este caso, cada bloque es de 1 KB.) que ocupa cada directorio (o cada fichero, si utilizamos la opción `-a`) pertenecientes al directorio que especifiquemos (o al directorio actual si no indicamos directorio):

```
petra:~/d1$ du .
24      ./d2
49      .
petra:~/d1$ du -a
1       ./f1
1       ./f1.log
2       ./f2
1       ./f3
18      ./fichero
1       ./forward
1       ./d2/p23
2       ./d2/hlog
1       ./d2/pie
18      ./d2/par
1       ./d2/carta
24      ./d2
49      .
petra:~/d1$
```

Podemos ver cómo en la primera llamada (sin opciones) nos muestra los bloques que ocupan tanto el directorio d2 (con todo lo que incluya) como el propio directorio actual (con ficheros y directorios que contenga). La segunda orden (con la opción `-a`) nos muestra además lo que ocupan cada uno de los ficheros.

2.2. Órdenes básicas de manejo de ficheros

Cuando un usuario entra en sesión el sistema le sitúa en un directorio propiedad del usuario. Este directorio se suele denominar el directorio HOME del usuario, y es la parte del disco donde el usuario tiene completa libertad crear y borrar ficheros y directorios.

La primera orden que puede ser útil es la orden `ls` (iniciales de *list sorted*, listado ordenado), que muestra por pantalla el contenido (nombre de los ficheros) que hay en el directorio actual.

```
petra:~/d1$ ls
d2  f1  f1.log  f2  f3  fichero  forward
petra:~/d1$
```

Como casi todas las ordenes de Unix, *ls* tiene multitud de opciones. Entre ellas podemos destacar:

- `l`: Listado largo. Muestra más información acerca de cada fichero además del nombre.
- `a`: muestra todos los ficheros, incluyendo los ocultos. Los ficheros cuyo nombre comienza por un punto (.) se suponen ocultos en Unix, y no aparecen en los listados normales.

```
petra:~/d1$ ls
d2  f1  f1.log  f2  f3  fichero  forward
petra:~/d1$ ls -a
.  ..  d2  f1  f1.log  f2  f3  fichero  forward
petra:~/d1$ ls -al
total 28
```

```
drwxr-xr-x    3 albizu   profes       1024 Oct 23 10:13 .
drwx--x--x   34 albizu   profes       2048 Oct 23 10:13 ..
drwxr-xr-x    2 albizu   profes       1024 Oct 23 10:14 d2
-rw-r--r--    1 albizu   profes         26 Oct 22 13:28 f1
-rw-r--r--    1 albizu   profes        237 Oct 23 10:12 f1.log
-rw-r--r--    1 albizu   profes       1393 Oct 16 16:46 f2
-rw-r--r--    1 albizu   profes         47 Oct 22 13:28 f3
-rw-------    1 albizu   profes      17225 Dec 15  1999 fichero
-rw-r--r--    1 albizu   profes         25 Dec 19  2000 forward
petra:~/d1$
```

Como puede observarse, las opciones en Unix se preceden de un guión, y en caso que queramos introducir más de una opción podemos juntar las opciones precediéndolas de un único guión.

Otras ordenes que nos pueden ser útiles son las siguientes:

- `cat fichero1 ...`: muestra por pantalla el contenido de los ficheros especificados.
- `cp fichero1 fichero2`: copia el contenido del *fichero1* en el *fichero2*, creándolo si no existía y machacando su contenido si ya existía.
- `mv fichero1 fichero2`: mueve el *fichero1* (eliminándolo) y llevando su contenido al *fichero2*. En el caso de que ambos ficheros estén en el mismo directorio, equivale a renombrar el fichero. También es aplicable a nombres de directorios.
- `rm fichero`: borra el *fichero* del disco. Una vez borrado el fichero éste no puede ser recuperado.

2.3. Órdenes relativos a directorios

Como ya hemos señalado anteriormente, cuando no indicamos un nombre de fichero completo (comenzando por / y nombrando todos los directorios por los que debemos pasar para llegar a él desde el raíz) estamos nombrando al fichero a partir del directorio actual o directorio de trabajo. En los sistemas Unix habituales no aparece en pantalla el directorio actual (como aparece en el *prompt* del Linux), con lo que cuando deseamos saber cuál es el directorio actual debemos introducir la orden *pwd (print working directory).*

```
petra:~/d1$ pwd
/home/profesores/albizu/d1
petra:~/d1$
```

El resto de las ordenes que hay en Unix para trabajar con directorios son las siguientes:

- `cd directorio`: cambia el directorio de trabajo al *directorio* especificado. Este directorio puede darse (como cualquier fichero) con el nombre completo (a partir del raíz) o a partir del directorio actual.
- `mkdir directorio`: crea el *directorio* especificado.
- `rmdir directorio`: borra el directorio indicado, siempre y cuando se encuentre vacío.

Asimismo, es aplicable la orden *mv* antes vista.

2.4. Propiedad de los ficheros

Otro aspecto importante en Unix es la propiedad de los ficheros. Al ser un sistema multiusuario, y para poder garantizar su protección, los ficheros en Unix pertenecen a un

usuario (propietario del fichero - *owner*) y a un grupo (*group*). Estos campos los habíamos visto anteriormente cuando examinábamos la salida de la orden *ls -l*, donde podemos apreciar cual es el propietario y el grupo al cual pertenece el propietario. Estos atributos del fichero se crean, por defecto, automáticamente con el nombre del usuario que crea el fichero y el grupo al cual pertenezca éste.

Hay un par de ordenes que nos permiten cambiar estos dos atributos de un fichero determinado. Sólo el propietario puede ejecutar estas ordenes, por razones obvias. Estas ordenes son las siguientes:

• `chown propietario fichero`: establece como nuevo *propietario* del *fichero* el indicado.

• `chgrp grupo fichero`: establece como nuevo *grupo* al cual pertenece el *fichero* el indicado.

2.5. Permisos de acceso a ficheros

Como ya se ha indicado, el Unix tiene un esquema de protección de ficheros para evitar accesos no deseados de usuarios no autorizados. Para ello, uno de los atributos que tiene asociado cada fichero son sus permisos de acceso. Anteriormente hemos visto como en la salida de la orden `ls -l` todos los ficheros iban acompañados por una cadena de caracteres (r, w, x, -), indicando los permisos de acceso a cada fichero.

El sistema Unix distingue tres tipos de accesos a ficheros:

- Acceso para lectura (*r*). En este tipo de accesos, no se modifica el contenido del fichero, sino que sólo se consulta.
- Acceso para escritura (*w*). En este caso, la información contenida en el fichero se va a modificar, bien porque se va a añadir información, se va a suprimir o se va a cambiar la información existente.
- Acceso para ejecución (*x*). El fichero en esta ocasión va a ser accedido para ser ejecutado. Solamente los ficheros en código ejecutable por la máquina y los ficheros de órdenes (*shell scripts*) pueden ser ejecutados, por lo que únicamente tiene sentido en esos ficheros realizar accesos para ejecución. En los directorios este tipo de acceso adquiere otro significado, como veremos en seguida.

Dado que no todos los usuarios son iguales en el sistema, parece lógico pensar que no todos los usuarios van a tener los mismos privilegios para acceder a todos los ficheros. Por lo tanto, los tres tipos de acceso a los ficheros se establecen a tres niveles distintos:

- Propietario del fichero.
- Grupo al que pertenece el fichero.
- Resto de los usuarios.

Para representar estos nueve niveles de protección distintos (tres tipos de acceso para tres tipos de usuarios), el sistema Unix utiliza una cadena de 9 caracteres, donde los 3 primeros indican los tipos de acceso permitidos para el usuario, los 3 siguientes los tipos de acceso para el grupo al que pertenece el fichero, mientras que los tres últimos hacen referencia al resto de los usuarios del sistema.

Cuando un tipo de acceso está permitido aparece una letra (r para el permiso de lectura, w para el de escritura y x para el de ejecución, y en este orden), mientras que si un acceso no está permitido aparece un guión (-) en el lugar correspondiente al permiso.

Así, en el siguiente caso:

```
petra:~/d1$ ls -l
total 25
drwxr-xr-x    2 albizu   profes       1024 Oct 23 10:14 d2
-rwxr--r--    1 albizu   profes         26 Oct 22 13:28 f1
-rw-r--r--    1 albizu   profes        237 Oct 23 10:12 f1.log
-rw-r--r--    1 albizu   profes       1393 Oct 16 16:46 f2
-rw-r--r--    1 albizu   profes         47 Oct 22 13:28 f3
-rw-------    1 albizu   profes      17225 Dec 15  1999 fichero
-rw-r--r--    1 albizu   profes         25 Dec 19  2000 forward
petra:~/d1$
```

El fichero f1 tiene establecidos unos permisos rwxr-xr--. Los permisos que se aplican la propietario (albizu en este caso) son los tres primeros: puede leer, escribir ejecutar el fichero. A cualquier usuario del grupo profes se le aplican los tres siguientes permisos (r-x): podrá leer y ejecutar el fichero. Cualquier otro usuario podrá sólo leer el fichero.

Como podemos ver, los permisos que se aplican son los que corresponden con el usuario. Es decir, al propietario del fichero le afectan los tres primeros permisos; a los usuarios del grupo al que pertenece el fichero (a excepción del propietario, en el caso de que pertenezca al grupo), los tres permisos intermedios; al resto de los usuarios (los que no sean el propietario ni del grupo al que pertenece el fichero) los tres últimos.

Modificación de permisos de acceso a ficheros: orden *chmod*

Para establecer los permisos de acceso a un fichero se utiliza la orden *chmod*. El formato de esta orden es el siguiente:

```
chmod permisos ficheros
```

Donde *ficheros* son los nombres de ficheros a los que se les quiere poner unos *permisos* de acceso determinados. Hay dos formas de especificar los permisos que queremos establecer para un conjunto de ficheros:

- Especificándolo mediante el formato `[ugo] [+-] [rwx]`. Mediante el primer conjunto de letras se especifica para quién se van a modificar los permisos: `u` para el propietario, `g` para el grupo y `o` para otros. El `+` o el `-` especifica si se va a dar o quitar el permiso. El último grupo de letras indica el permiso a modificar: `r` para leer, `w` para escribir y `x` para ejecutar. Se puede introducir cualquier combinación de las letras relativas tanto a los usuarios como a los permisos. Así, podría indicar, por ejemplo, `go-wx` para indicar que se quitaran los permisos de escritura y de ejecución al grupo y a otros.
- Especificándolo mediante un número octal de tres cifras. La primera cifra hace referencia a los permisos relativos al propietario, la segunda al grupo y la tercera a el resto de usuarios. Cada uno de las cifras en octal se forma sumando las cifras correspondientes a cada permiso que queremos conceder: 4 para lectura, 2 para escritura y 1 para ejecución. Así, si queremos establecer unos permisos rwxr-xr— deberemos indicar 754: 7=4(por la r)+2(por la w)+1(por la x), 5=4(por la r)+ 1(por la x), 4=4(por la r).

```
petra:~/ejemplo$ ls -l
total 3
d--x------    2 albizu   profes       1024 Oct 23 10:37 ejecutar
-rw-r--r--    1 albizu   profes          1 Oct 25 10:15 f
dr--------    2 albizu   profes       1024 Oct 23 10:37 leer
petra:~/ejemplo$ chmod go-r f
petra:~/ejemplo$ ls -l
total 3
d--x------    2 albizu   profes       1024 Oct 23 10:37 ejecutar
-rw-------    1 albizu   profes          1 Oct 25 10:15 f
dr--------    2 albizu   profes       1024 Oct 23 10:37 leer
petra:~/ejemplo$ chmod 744 f
```

```
petra:~/ejemplo$ ls -l
total 3
d--x------    2 albizu   profes       1024 Oct 23 10:37 ejecutar
-rwxr--r--    1 albizu   profes          1 Oct 25 10:15 f
dr--------    2 albizu   profes       1024 Oct 23 10:37 leer
petra:~/ejemplo$
```

Vemos en el primer caso como quitamos los permisos de lectura para el grupo y para el resto de usuarios al fichero f, mientras que en el segundo caso damos todos los permisos al propietario y sólo el de lectura al grupo y a otros usuarios.

Modificación de la máscara de creación de ficheros: orden *umask*

Cuando creamos un fichero el sistema le asigna unos permisos por defecto. Si queremos cambiarlos podemos utilizar la orden anterior. Pero si no nos gustan los permisos que asigna el sistema por defecto sería muy incomodo tener que andar cambiando los permisos de cada fichero que creamos. Para evitar eso podemos modificar los permisos por defecto que asigna el sistema. Lo podemos hacer por medio de la orden *umask*. Su formato es:

```
umask permisos
```

Lo que hace esta orden es establecer lo que se denomina la *máscara* de creación de ficheros, es decir, los permisos que se asignarán a los ficheros que se creen. La forma de especificar los permisos es similar al caso anterior:

- Utilizando la combinación de las letras `[ugo][+-][rwx]` indicamos qué permisos queremos añadir o quitar a la máscara de creación de ficheros.
- Utilizando un número en octal indicamos específicamente la máscara que queremos establecer.

Si introducimos la orden sin especificar los permisos nos indica la máscara actual:

```
petra:~/ejemplo$ umask
022
petra:~/ejemplo$
```

Lo que indica la máscara es, siguiendo el formato octal antes indicado, los permisos que se van a quitar a los ficheros creados. Así, en este caso, nos dice que al propietario no se le va a quitar ningún permiso, al grupo el permiso de escritura y a otros el permiso de escritura.

Una consideración que hay que hacer es que, independientemente de la máscara que se establezca, cuando se crea un fichero no se crea con permiso de ejecución. Este permiso hay que establecerlo manualmente. Si se crean directorios sí se crean con permiso de ejecución, si es que así lo contempla la máscara.

Si queremos modificarlos podemos utilizar la orden umask con cualquiera de los formatos antes vistos, tanto con el número en octal como con la combinación de letras `ugo+-rwx`.

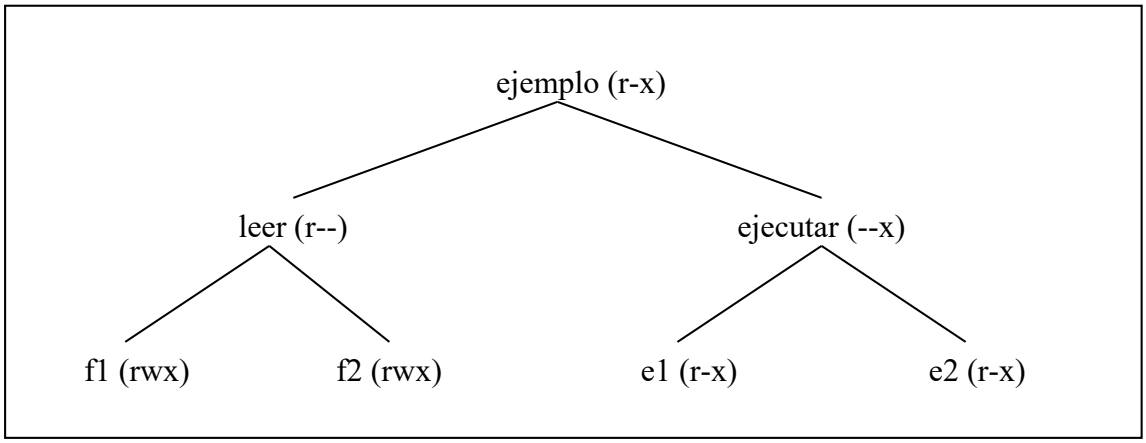
```
petra:~/ejemplo$ umask
022
petra:~/ejemplo$ echo >f1
petra:~/ejemplo$ ls -l
total 4
d--x------    2 albizu   profes       1024 Oct 23 10:37 ejecutar
-rwxr--r--    1 albizu   profes          1 Oct 25 10:15 f
-rw-r--r--    1 albizu   profes          1 Oct 25 11:07 f1
dr--------    2 albizu   profes       1024 Oct 23 10:37 leer
petra:~/ejemplo$ umask go-r
petra:~/ejemplo$ umask
066
petra:~/ejemplo$ echo >f2
```

```
petra:~/ejemplo$ ls -l
total 5
d--x------    2 albizu   profes       1024 Oct 23 10:37 ejecutar
-rwxr--r--    1 albizu   profes          1 Oct 25 10:15 f
-rw-r--r--    1 albizu   profes          1 Oct 25 11:07 f1
-rw-------    1 albizu   profes          1 Oct 25 11:07 f2
dr--------    2 albizu   profes       1024 Oct 23 10:37 leer
petra:~/ejemplo$
```

El permiso de ejecución en directorios

Ya hemos visto que sólo parece tener sentido el permiso de ejecución en ficheros susceptibles de ser ejecutados. En ningún caso los directorios pueden ser ejecutados, pero se utiliza también el permiso de ejecución en este caso, si bien con un significado distinto.

El permiso de ejecución para el caso de directorios podemos traducirlo como "permiso de acceso o de búsqueda" al directorio. Es frecuente confundir ambos tipos de acceso a los directorios, pero son claramente distintos. En el caso de tener acceso para lectura a un directorio lo único que se nos garantiza es el poder conocer qué ficheros contiene ("hacer un ls sobre el directorio"), mientras que si tenemos permisos de ejecución podremos acceder a los ficheros del directorio, así como poder situar el directorio actual en él. Por ejemplo, supongamos el siguiente árbol de directorios, donde los rectángulos indican directorios y los óvalos ficheros, y donde los permisos que se indican entre paréntesis son los efectivos del usuario que intenta acceder (los 3 primeros si es el propietario, los tres siguientes si es del grupo al que pertenece el fichero o los tres últimos en otro caso):



Estando situados en el directorio ejemplo, podremos consultar el contenido del directorio leer, pero no se podrá hacer ninguna operación sobre los ficheros en él contenidos, a pesar de tener permisos para ello.

```
petra:~/ejemplo$ ls
ejecutar  leer
petra:~/ejemplo$ ls leer
f1  f2
petra:~/ejemplo$ cat leer/f1
cat: leer/f1: Permission denied
```

Sin embargo, sobre el directorio ejecutar, a pesar de no poder realizar el *ls*, pero si podremos realizar la orden

```
petra:~/ejemplo$ ls ejecutar
ls: ejecutar: Permission denied
petra:~/ejemplo$ cat ejecutar/e1
echo Soy el fichero e1
petra:~/ejemplo$
```

¡Si es que sabemos que está ahí!

2.6. Enlaces simbólicos

Examinando los ficheros que hay en el disco, a veces aparecen ficheros con una l como primera letra en el campo de permisos.

```
petra:/$ ls -l
total 99
drwxr-xr-x    2 root     root         2048 Apr 27 23:57 bin
drwxr-xr-x    2 root     root         1024 Sep 14  2000 boot
drwxr-xr-x    2 root     root         1024 May 15  2000 cdrom
drwxr-xr-x    3 root     root        19456 Oct 23 06:35 dev
drwxr-xr-x   83 root     root         5120 Oct 23 10:39 etc
drwxr-xr-x    2 root     root         1024 May 15  2000 floppy
drwxr-xr-x   13 root     root         1024 Jul  6 12:42 home
drwxr-xr-x    2 root     root         1024 May 15  2000 initrd
drwxr-xr-x    4 root     root         4096 May 28 13:52 lib
drwxr-xr-x    2 root     root        12288 Mar 10  1999 lost+found
drwxr-xr-x    4 root     root         1024 Feb  2  2000 mnt
dr-xr-xr-x   91 root     root            0 Sep 24 09:10 proc
drwx------    5 root     root        45056 Oct 22 14:56 root
drwxr-xr-x    2 root     root         3072 Dec 18  2000 sbin
lrwxrwxrwx    1 root     root            8 Mar 13  2000 tmp -> /var/tmp
drwxr-xr-x   15 root     root         1024 May 15  2000 usr
drwxr-xr-x   18 root     root         1024 Jan 23  2001 var
lrwxrwxrwx    1 root     root           20 Jun 12  2000 vmlinuz -> /boot/vmlinuz
-2.2.16
petra:/$
```

Los nombres donde aparece esa l en realidad no son ficheros: son enlaces simbólicos. Los enlaces simbólicos son simplemente un nombre alternativo que se puede crear para referenciar un fichero o un directorio. Así podemos acceder al mismo fichero por cualquiera de sus nombres: el "original" y cualquiera de los enlaces que hayamos creado. Se puede ver en lugar del nombre del fichero el nombre del enlace y una flecha indicando cuál es fichero original.

Se pueden crear enlaces tanto de ficheros como de directorios, pudiendo estar los enlaces en el mismo o en distinto directorio que el fichero original.

Para crear un enlace existe la orden *ln* (de *link*, enlace), con la opción *-s* (por simbólico):

```
ln -s fichero nombreEnlace
```

Esta orden crearía un enlace con el nombreEnlace especificado al fichero indicado.

Para borrar un enlace simbólico, independientemente que sea a un fichero a un directorio, se utiliza la orden `rm` antes vista.

2.7. Caracteres comodín

En muchas órdenes es posible hacer referencia a varios ficheros, de manera que la orden afecta a todos ellos. Por ejemplo, si deseamos ver en detalle las propiedades (permisos, tipo de fichero, propietario, etc) de los ficheros que se denominen f1, f2 y f3 podemos introducir la orden:

```
petra:~$ ls -l f1 f2 f3
-rw-r--r--    1 albizu   profes         26 Oct 22 13:28 f1
-rw-r--r--    1 albizu   profes       1393 Oct 16 16:46 f2
-rw-r--r--    1 albizu   profes         47 Oct 22 13:28 f3
petra:~$
```

A menudo ocurre que cuando queremos utilizar una orden con más de un fichero los nombres de éstos son parecidos: comienzan por la misma letra, terminan igual, etc. Para poder hacer referencia a todos los nombres de ficheros que compartan algo en común se

pueden utilizar los *caracteres comodines*, o las *expresiones regulares* (los caracteres comodines son un caso concreto de expresiones regulares).

Cuando confeccionamos un nombre de fichero en Unix podemos utilizar una expresión regular, que podemos considerarla como un patrón. La expresión regular será sustituida por el shell por el conjunto de ficheros que se ajusten al patrón, pasando a la orden correspondiente el conjunto de nombres de ficheros en lugar de la expresión regular. Los caracteres que podemos utilizar para construir la expresión regular son los siguientes:

- `?` : Cuadra con cualquier carácter, pero sólo con uno.
- `*` : Cuadra con cualquier carácter y con cualquier número de caracteres.
- `[caracteres]`: Cuadra con cualquiera de los caracteres introducidos entre los corchetes.
- `[car1-car2]`: Cuadra con cualquiera de los caracteres que estén entre car1 y car2
- `[!caracteres]`: Cuadra con cualquier carácter menos con los que están entre los corchetes.

La expresión regular se puede formar con la combinación de algunos o con todos los caracteres comodín

Veamos algunos ejemplos.

- La expresión `f?` se ajusta a todos los nombres de ficheros que existan que comiencen con f y tengan 2 caracteres. Por ejemplo, f1, fa, f- cuadrarían con la expresión, pero f, f12 no lo hacen.
- La expresión `f*` se ajusta a todos los nombres que empiecen por f. Los ficheros f, f1, f12, …, cuadran con la expresión. La única condición es que comiencen por f.
- La expresión `??*` cuadra con todos los nombres de ficheros que tengan al menos 2 caracteres.
- La expresión `[abc]*` cuadra con todos los nombres de ficheros que comiencen por a, b o c.
- La expresión `[!0-9]?*` cuadra con todos los nombres de ficheros que no comiencen por cualquier carácter (a excepción de un dígito) y que al menos tenga 2 caracteres de longitud.

La utilización de expresiones regulares para hacer referencia a más de un fichero puede utilizarse con cualquier orden que acepte varios ficheros como parámetros, que son la mayoría de las órdenes de manejo de ficheros. Las órdenes de copiado y movimiento de ficheros (cp, mv) aceptan múltiples ficheros como parámetros, pero en este caso el último parámetro debe ser un directorio a donde copiar o mover los múltiples ficheros.

Ejercicios

1. Vete al directorio raíz y utiliza la orden `cd` sin parámetros para volver a tu directorio de trabajo.
2. Situarse en el directorio `/etc` y acceder a uno de los ficheros situados en el directorio de trabajo propio utilizando el camino, ruta de acceso o pathname de las dos formas posibles: absoluta y relativa.
3. Situarse en el directorio raíz. Comprobar qué directorios existen en éste. Comprobar que existen directorios.
4. Comprobar el tipo de los ficheros de un directorio determinado, por ejemplo del raíz.
5. Ir al directorio `/dev` y comprobar que contiene ficheros de tipo dispositivo. Comprobar también que se da información acerca del número mayor y menor justo antes de la fecha de modificación.
6. Crear un fichero con dos nombres diferentes.
7. Comprobar mediante la acción anterior que aparece un 2 en la segunda columna, indicando con ello que existen dos nombres para el mismo fichero.
8. Comprobar que efectivamente se trata de un sólo fichero, modificando desde uno de ellos y comprobando desde el otro que ha sido modificado.
9. Crear un nuevo enlace al mismo fichero y comprobar que se ha creado.
10. Situarse en el directorio `HOME` y comprobar los permisos sobre los ficheros existentes.
11. Crear un fichero de prueba para modificarle los permisos, por ejemplo que contenga el listado de los ficheros del directorio.
12. Comprobar los permisos actuales y hacer que sólo el propietario tenga permisos de cualquier tipo.
13. Añadir ahora permisos de escritura para el grupo.
14. Crear un directorio de prueba si no existe.
15. Modificar los permisos de éste para que sólo sean de lectura por parte del propietario. Comprobar que se puede leer su contenido, pero no se puede entrar en él ni crear en él nuevos ficheros.
16. Crear situaciones similares con sólo permisos de ejecución.
17. Cambiar el propietario de un fichero. Ojo, que después el nuevo propietario es el que tiene capacidad para borrarlo.
18. Inspeccionar el contenido de algunos ficheros. Pueden ser interesantes ficheros como el `/etc/passwd` y el `/etc/group`. Averigua, a partir de los ficheros anteriores, el número de usuario, número de grupo y nombre de grupo correspondiente a cada usuario.
19. Buscar mediante la ayuda los tipos de ficheros que puedan aparecer como primer campo del `ls`.
20. Indique la expresión regular que hay que introducir para referirnos a todos los ficheros cuyo nombre comience por "a" y terminen por un dígito.

21. Indique la expresión regular que hay que introducir para referirnos a todos los ficheros cuyo nombre contenga tres caracteres, siendo el primero necesariamente una letra (mayúscula o minúscula).
22. Indique la expresión regular que hay que introducir para referirnos a todos los ficheros cuyo nombre contenga un "." en cualquier posición a excepción de la primera.
23. Indique la expresión regular que hay que introducir para referirnos a todos los ficheros cuyo nombre no termine por ".c".
24. Indique la expresión regular que hay que introducir para referirnos a todos los ficheros cuyo nombre tenga al menos 2 caracteres.
25. Inspeccionando los ficheros que tengo en mi directorio me he encontrado con que tengo uno que se llama "." y otro "..". Yo no los he creado y no se lo que significan. ¿Quién los ha creado, y qué utilidad tienen?
26. ¿Para qué sirve la orden `df`? ¿Cómo se utiliza?
27. ¿Para qué sirve la orden `mount`? ¿Cómo se utiliza?
28. ¿Para qué sirve la orden `quota`? ¿Cómo se utiliza?
29. ¿Qué permisos necesito tener para poder crear un fichero? Indique todos aquellos que sean necesarios, tanto del fichero como del directorio al que pertenezca.
30. ¿Qué permisos necesito tener para poder ejecutar un fichero (se supone que contiene código ejecutable)? Indique todos aquellos que sean necesarios, tanto del fichero como del directorio al que pertenezca.
31. ¿Qué permisos necesito tener para poder eliminar un fichero? Indique todos aquellos que sean necesarios, tanto del fichero como del directorio al que pertenezca.
32. ¿Qué permisos necesito tener para poder leer un fichero? Indique todos aquellos que sean necesarios, tanto del fichero como del directorio al que pertenezca.
33. ¿Qué permisos necesito tener para poder modificar un fichero? Indique todos aquellos que sean necesarios, tanto del fichero como del directorio al que pertenezca.
34. Quiero borrar todos los ficheros del directorio actual. Suponiendo que tengo los permisos necesarios, ¿cómo puedo hacerlo?
35. Quiero borrar un directorio "`d1`", perteneciente al directorio actual, y todo su contenido. Suponiendo que tengo los permisos necesarios, ¿Cómo puedo hacerlo mediante una única orden?
36. Quiero cambiar el directorio actual a un "hermano" (hijo del mismo padre) del actual. El directorio al que quiero cambiar se llama "`d1`". ¿Cómo se hace con una única orden?
37. Quiero conocer la mayor información posible (permisos, tamaño, etc.) del directorio actual. ¿Cómo puedo hacerlo mediante una única orden?
38. Quiero conocer la mayor información posible (permisos, tamaño, etc.) del directorio padre del actual. ¿Cómo puedo hacerlo mediante una única orden?
39. Quiero copiar todos los ficheros del directorio actual a un directorio que se llama /tmp. Suponiendo que tengo los permisos necesarios, ¿Cómo puedo hacerlo?

40. Quiero crear un directorio hijo del directorio "`d1`", que es hijo del directorio actual. Indique cómo se hace con una única orden
41. Quiero mover todos los ficheros del directorio actual a un directorio que se llama `/tmp`. Suponiendo que tengo los permisos necesarios, ¿Cómo puedo hacerlo?

3. Órdenes relativas a procesos y usuarios

El sistema operativo Unix está estructurado en torno a dos conceptos: al de fichero y al de proceso. Esta estructura también se ve reflejada en las órdenes más habituales que se utilizan. También conocemos que el sistema Unix es un sistema multiusuario, donde puede haber varios usuarios ejecutando procesos simultáneamente. En este apartado veremos las órdenes más utilizadas relacionadas con procesos y con usuarios.

3.1. Obtención de información sobre usuarios y sobre el sistema

Es frecuente la utilización de órdenes para saber quién está conectado al sistema en un momento dado y qué está haciendo. Las órdenes que más se utilizan para ese fin son las siguientes:

- `uname`: Muestra información sobre el sistema al que estamos conectados. Con la opción *–a* muestra toda la información relevante: nombre del sistema operativo, nombre de la máquina, versión, etc.

```
petra:~$ uname -a
Linux petra 2.2.16 #1 SMP Mon Jun 12 09:48:05 CEST 2000 i686 unknown
petra:~$
```

- `who`: muestra el nombre de usuario de todos los que se encuentran conectados en este momento. Un caso particular de esta orden es *who am i*, que muestra nuestro nombre.

```
petra:~$ who
i0214904 pts/0    Oct 29 11:06 (pcinf040.euitio.uniovi.es)
i9429000 pts/2    Oct 29 10:39 (pcinf043.euitio.uniovi.es)
albizu   pts/5    Oct 29 11:07 (pcpio01.ccu.uniovi.es)
i5434261 pts/6    Oct 29 10:58 (pet13.geol.uniovi.es)
i1443237 pts/7    Oct 29 11:14 (pcinf057.euitio.uniovi.es)
i2128593 pts/10   Oct 29 11:15 (pcinf055.euitio.uniovi.es)
i1645561 pts/14   Oct 29 11:15 (213.98.110.26)
nestor   pts/16   Oct 29 09:42 (pcpio50.ccu.uniovi.es)
i9436428 pts/17   Oct 29 11:04 (pcinf019.euitio.uniovi.es)
secr     pts/18   Oct 29 10:09 (dir03.ccu.uniovi.es)
i6945947 pts/19   Oct 29 10:09 (pcbio14.bio.uniovi.es)
petra:~$ who am i
petra!albizu   pts/5    Oct 29 11:07 (pcpio01.ccu.uniovi.es)
petra:~$
```

- `whoami`: muestra el nombre del usuario.
- `logname`: similar al anterior, pero muestra el nombre del usuario que realizó la conexión al sistema. Habitualmente es igual al mostrado por la orden anterior, pero en algún caso, como veremos, puede cambiar.
- `id`: muestra para el usuario activo su nombre, el grupo activo y los grupos a los que pertenece.

```
petra:~$ whoami
albizu
```

```
petra:~$ logname
albizu
petra:~$ id
uid=1004(albizu) gid=1000(profes) groups=1000(profes)
petra:~$
```

- `finger`: similar al anterior, pero muestra además el nombre real de cada usuario conectado, si es que esa información consta en el sistema. Poniendo como parámetro un nombre de usuario, nos muestra la información que contiene el sistema sobre ese usuario, aunque no esté conectado.

```
petra:~$ finger
Login    Name               Tty      Idle  Login Time   Office     Office Phone
albizu.  Miguel Riesco Alb *pts/5         Oct 29 11:07 (pcpio01.ccu.uniovi.es)
angel    Angel Luis Lopez  *pts/1         Oct 29 11:15 (213.97.185.123)
i1000157 Carlos Gordo Obes  pts/9       1  Oct 29 11:14 (pcinf054.euitio.uniovi.es)
i1122332 Carlos Garcia Rodr  pts/10        Oct 29 11:15 (pcinf055.euitio.uniovi.es)
i5434261 Daniel Alejo Fern *pts/6      13  Oct 29 10:58 (pet13.geol.uniovi.es)
i6945947 Javier Alvarez Lo *pts/19      8  Oct 29 10:09 (pcbio14.bio.uniovi.es)
i6948635 ANDRES FERNANDEZ  *pts/3          Oct 29 11:15 (pcinf039.euitio.uniovi.es)
i9429000 Vanessa Prieto Go  pts/2          Oct 29 10:39 (pcinf043.euitio.uniovi.es)
petra:~$ finger albizu
Login: albizu                           Name: Miguel Riesco Albizu
Directory: /home/profesores/albizu      Shell: /bin/bash
Office: 202, 985103390                  Home Phone: 12323123123123
On since Mon Oct 29 11:07 (CET) on pts/5 from pcpio01.ccu.uniovi.es (messages off)
No mail.
Plan:

Login: zubi                             Name: Miguel Riesco Albizu
Directory: /home/profesores/zubi        Shell: /bin/bash
Last login Mon Sep 10 10:52 (CEST) on pts/12 from pcpio01.ccu.uniovi.es
Mail forwarded to albizu@lsi.uniovi.es
No mail.
No Plan.
petra:~$
```

- `chfn`: permite a los usuarios cambiar la información que consta en el sistema sobre ellos.

- `w`: similar al *who*, pero muestra también qué está ejecutando cada usuario en ese momento.

```
petra:~$ w
 11:17am  up 35 days,  3:05, 15 users,  load average: 0.13, 0.18, 0.17
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
i9429000 pts/2    pcinf043.euitio. 10:39am 43.00s  0.33s  0.02s  man vmstat
albizu   pts/5    pcpio01.ccu.unio 11:07am  1.00s  0.31s  0.10s  w
i5434261 pts/6    pet13.geol.uniov 10:58am 14:27   0.28s  0.15s  pine -i
i6951630 pts/7    pcinf042.euitio. 11:17am  4.00s  0.26s  0.04s  pine
i2128593 pts/10   pcinf055.euitio. 11:15am  1.00s  0.20s  0.20s  -bash
i1888157 pts/9    pcinf054.euitio. 11:14am  2:50   0.58s  0.44s  pine
i1644866 pts/11   fym01.ciencias.u 10:54am  3:18   0.69s  0.57s  pine
i1636962 pts/13   econo004.econo.u  9:21am  1:53m  1.18s  1.10s  pine
nestor   pts/16   pcpio50.ccu.unio  9:42am  1:35m  0.02s  0.02s  -bash
i9436428 pts/17   pcinf019.euitio. 11:04am  5:18   0.17s  0.04s  pine
secr     pts/18   dir03.ccu.uniovi 10:09am  1:18m  0.02s  0.02s  -bash
i6945947 pts/19   pcbio14.bio.unio 10:09am  9:33   0.22s  0.10s  pine
petra:~$
```

3.2. Órdenes relacionadas con procesos

Hay varias órdenes que se utilizan a menudo relacionadas con procesos. Las más útiles pueden ser las siguientes:

- `ps`*: (processes status)* muestra la lista de procesos que hay en el sistema. Sin opciones sólo muestra los que tiene el usuario en esa sesión. Tiene multitud de opciones para mostrar más información (*l, x)*, para mostrar todos los procesos del

sistema (*a)*, para incluir información del usuario al que pertenece cada proceso (*u*), etc.

```
petra:~$ ps
  PID TTY          TIME CMD
 5003 pts/5    00:00:00 bash
 5692 pts/5    00:00:00 ps
petra:~$ ps ux
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
albizu    5003  0.0  0.1  1980 1180 pts/5    S    11:07   0:00 -bash
albizu    5693  0.0  0.1  2636  976 pts/5    R    11:24   0:00 ps ux
petra:~$
```

La información que presenta en cada caso varía de los modificadores que se incluyan, pero siempre muestra el PID (identificador del proceso) y el COMMAND (programa que está ejecutando).

- `top`*:* de *table of* processes, muestra un conjunto de procesos (los que caben en pantalla) que están en ejecución en este momento. Esta orden va actualizando la lista repetidamente, hasta que salimos. Es un programa interactivo que tiene varias opciones que nos permiten hacer cosas diversas con los procesos que estamos viendo.

```
11:29am  up 35 days,  3:17, 14 users,  load average: 0.19, 0.20, 0.18
81 processes: 78 sleeping, 1 running, 0 zombie, 2 stopped
CPU states:  0.3% user,  1.3% system,  0.0% nice, 98.1% idle
Mem:  971868K av, 944368K used,  27500K free,  60316K shrd, 633304K buff
Swap: 130748K av,    692K used, 130056K free                200788K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
 5890 albizu    10   0  1136 1136   684 R       0  1.9  0.1   0:00 top
 2985 www-data   1   0  2204 2136  1992 S       0  0.7  0.2   0:00 apache
 5891 telnetd    8   0   560  560   456 S       0  0.3  0.0   0:00 in.telnetd
 5892 root       8   0   708  708   548 S       0  0.3  0.0   0:00 login
    4 root       0   0     0    0     0 SW      0  0.0  0.0   0:00 kpiod
    5 root       0   0     0    0     0 SW      0  0.0  0.0   2:02 kswapd
   82 daemon     0   0   336  320   252 S       0  0.0  0.0   0:00 portmap
  147 root       0   0   604  600   488 S       0  0.0  0.0  12:58 syslogd
  149 root       0   0   612  604   324 S       0  0.0  0.0   0:00 klogdd
```

- `uptime`: muestra el tiempo que lleva la máquina arrancada y un indicador de la carga de la máquina. La carga se mide por número medio de procesos listos para ejecución en un periodo de tiempo determinado, mostrándose la carga en el último minuto, en los últimos 5 minutos y en los últimos 15 minutos.

```
petra:~$ uptime
 11:30am  up 35 days,  3:18, 15 users,  load average: 0.34, 0.23, 0.19
petra:~$
```

- `kill`*:* se utiliza para mandar señales a procesos. Su utilidad más habitual es para mandar la señal número 9, que fuerza la terminación del proceso. En el caso que no se indique señal, se envía la 15 (terminación de proceso) El proceso se identifica pasando como parámetro el identificador del proceso (*PID).*

```
petra:~$ ps
  PID TTY          TIME CMD
 5003 pts/5    00:00:00 bash
 6006 pts/5    00:00:01 prog
 6008 pts/5    00:00:00 ps
petra:~$ kill -9 6006
[1]+  Killed                  prog
petra:~$ ps
  PID TTY          TIME CMD
 5003 pts/5    00:00:00 bash
 6013 pts/5    00:00:00 ps
```

```
petra:~$
```

- `killall`: similar a la anterior, pero se identifica al proceso por su nombre, no por su número de PID.

```
petra:~$ ps
  PID TTY          TIME CMD
 5003 pts/5    00:00:00 bash
 6030 pts/5    00:00:01 prog
 6032 pts/5    00:00:00 ps
petra:~$ killall -9 prog
[1]+  Killed                  prog
petra:~$ ps
  PID TTY          TIME CMD
 5003 pts/5    00:00:00 bash
 6037 pts/5    00:00:00 ps
petra:~$
```

- `slay`: mata a todos los procesos pertenecientes a un usuario determinado. Esta orden, lógicamente, sólo la pueden ejecutar usuarios con los debidos privilegios.

3.3. Cambio de usuario o grupo

Un usuario en Unix tiene asociado un nombre de usuario y un grupo al que pertenece. Estas dos propiedades sirven al sistema para (entre otras cosas), junto con los permisos de cada fichero, determinar los distintos tipos de acceso que el usuario puede realizar a cada fichero.

Si un usuario dispone de varias cuentas puede cambiar su "personalidad" mediante la ejecución de la orden `su usuario`. A partir de ese momento la identidad del usuario que está trabajando pasará a ser la indicada, con lo que para todas las cuestiones relacionadas con la identidad del usuario se tendrá en cuenta esta nueva.

```
petra:~$ whoami
albizu
petra:~$ logname
albizu
petra:~$ su zubi
Password:
petra:/home/profesores/albizu$ whoami
zubi
petra:/home/profesores/albizu$ logname
albizu
petra:/home/profesores/albizu$
```

Respecto al grupo de usuario, un usuario puede pertenecer a varios grupos a la vez. Pero en un momento dado sólo está activo uno, que es el que se aplica para todo lo relacionado con el grupo al que pertenece el usuario. No obstante, el usuario puede cambiar libremente su grupo activo por medio de la orden *newgrp grupo*. Si el usuario pertenece al grupo a partir de ese momento el sistema tomará ese grupo como el del usuario a todos los efectos.

3.4. Ejecución en segundo plano

Unix es un sistema multitarea, con lo que un usuario puede ejecutar varios procesos simultáneamente. Sin embargo, todas las órdenes que hemos visto hasta ahora "se apropian" de la terminal, no dejándonos lanzar otro proceso hasta que terminan. Si queremos lanzar varios procesos, debemos lanzar algunos en *segundo plano*, de tal manera que no se apropian de la terminal, dejando el shell disponible para la introducción de más órdenes. Para ello, sólo tenemos que añadir un & tras el nombre de la orden: esto hará que se ejecute en segundo plano:

```
petra:~$ prog &
[1] 7252
petra:~$ ps
  PID TTY          TIME CMD
 6520 pts/21   00:00:00 bash
 7252 pts/21   00:00:03 prog
 7255 pts/21   00:00:00 ps
petra:~$
```

Vemos cómo tras introducir la orden el sistema nos muestra un mensaje, indicando que el proceso tiene el PID 7252 en este caso. Inmediatamente se muestra el prompt del sistema y podemos ejecutar más órdenes. En el ejemplo hemos introducido la orden `ps`, que se ejecuta en primer plano (tiene el control de la terminal), y nos muestra los procesos que hay en ejecución, donde vemos el proceso de PID 7252 ejecutando el programa `prog`.

En ocasiones lanzamos un proceso en primer plano y, ante su duración, decidimos que hubiera sido conveniente lanzarlo en segundo plano, para poder seguir haciendo cosas en primer plano. Para poder hacer eso hay que hacer dos cosas: primero parar el proceso (parar, no terminar) y después reanudarlo en segundo plano. Lo segundo es sencillo: sólo hay que ejecutar la orden `bg`, indicando el trabajo que queremos reanudar en segundo plano. Si hay varios parados, debemos ver cuál queremos reanudar por medio de la orden `jobs`, indicando en la orden `bg` el número de trabajo que antecede al programa que queremos reanudar.

Para parar un proceso tenemos dos posibilidades:

- Enviarle la señal 19 (SIGSTOP) al proceso. Esto podemos hacerlo si disponemos de otra terminal para ejecutar la orden kill o killall o el proceso está ejecutándose en segundo plano.
- Si el proceso está en primer plano, desde la terminal que tenga asociado podemos pulsar la tecla que esté definida como de suspensión de procesos. Habitualmente no está definida, pero podemos hacerlo con la orden `stty`.

Veamos un ejemplo. Lo primero que haremos será definir una tecla para que realice la función de suspensión del proceso en primer plano. Definiremos una tecla (por ejemplo, `Control-p`) para ese fin:

```
petra:~$ stty susp ^P
petra:~$ stty
speed 9600 baud; line = 0;
susp = ^P;
-brkint -imaxbel
petra:~$
```

Para introducir la tecla control-p en la línea de órdenes, hay que introducir la secuencia `ctl-v ctl-p`, dado que si no el shell intentaría interpretar el carácter control `-p` y ejecutar la orden que tuviera asociada (al no tener nada asociado no haría nada).

Vemos al ejecutar `stty` sin parámetros que la tecla ha sido definida.

Una vez hecho esto, podemos lanzar cualquier proceso y detenerlo con la tecla definida:

```
petra:~$ prog
(Pulsamos ctl-p)
[1]+  Stopped                 prog
petra:~$ ps x
  PID TTY      STAT   TIME COMMAND
 6520 pts/21   S      0:00 -bash
 7747 pts/5    S      0:00 -bash
 7931 pts/21   T      0:01 prog
 7932 pts/21   R      0:00 ps x
petra:~$ bg
[1]+ prog &
petra:~$ ps x
  PID TTY      STAT   TIME COMMAND
 6520 pts/21   S      0:00 -bash
 7747 pts/5    S      0:00 -bash
 7931 pts/21   R      0:04 prog
 7935 pts/21   R      0:00 ps x
petra:~$
```

Como vemos, el programa se detiene y aparece como parado (T indica parado o realizando una traza); más tarde lo reanudamos con bg y el proceso continúa su ejecución, como vemos en la orden ps posterior, pero sin apropiarse de la terminal, como muestra el hecho que aparezca de inmediato el prompt del sistema.

Si queremos realizar lo contrario (pasar de segundo plano a primer plano), el camino es similar: paramos el proceso (en este caso mediante el envío de la señal 19) y lo reanudamos, esta vez con la orden `fg`:

```
petra:~$ prog &
[1] 8092
petra:~$ jobs
[1]+  Running                 prog &
petra:~$ ps x
  PID TTY      STAT   TIME COMMAND
 6520 pts/21   S      0:00 -bash
 7747 pts/5    S      0:00 -bash
 8092 pts/21   R      0:01 prog
 8094 pts/21   R      0:00 ps x
petra:~$ kill -19 8092

[1]+  Stopped                 prog
petra:~$ ps x
  PID TTY      STAT   TIME COMMAND
 6520 pts/21   S      0:00 -bash
 7747 pts/5    S      0:00 -bash
 8092 pts/21   T      0:08 prog
 8109 pts/21   R      0:00 ps x
petra:~$ fg
prog
```

Vemos cómo tras lanzar el proceso en segundo plano ejecutamos la orden `jobs` y `ps` para verificar que en realidad se está ejecutando. Mandamos la señal 19 para pararlo y a continuación lo reanudamos en primer plano. Obviamente, en este caso no sale el prompt del sistema hasta que el proceso termine.

3.5. Prioridades de ejecución: orden nice

La planificación de procesos que utiliza Unix se basa en un esquema de prioridades. Los procesos que sean más importantes se ejecutarán más frecuentemente que los que tengan una prioridad más baja.

Cuando se quiere ejecutar un proceso que va a tardar mucho en terminar es importante hacerlo de manera que incida lo menos posible en el trabajo del resto de los usuarios. Por eso, es posible lanzar procesos indicando una prioridad más baja de la normal. Así el

proceso se ejecutará sólo cuando el resto de los procesos normales no necesiten hacerlo. En los sistemas interactivos esto apenas su pone retraso para el proceso, dado que la mayor parte del tiempo la CPU está inactiva al estar condicionada la ejecución de los procesos por las entradas que suministra interactivamente el usuario (puede verse en cualquier momento la carga del sistema con la orden `uptime` y comprobar cómo habitualmente la carga no es grande).

3.6. Temporización y planificación de la ejecución de órdenes

En ocasiones puede ser interesante retrasar la ejecución de órdenes durante un tiempo en espera de que ocurra algo, como que la carga del sistema disminuya, que llegue una hora determinada, etc.

Para ello existen una serie de órdenes que permiten esperar un tiempo determinado, ejecutar órdenes a una hora determinada, ejecutar órdenes cuando la carga del sistema sea baja y planificar la ejecución de órdenes periódicamente.

1.1.1 La orden *sleep*

Esta orden espera durante la cantidad de segundos que se especifique sin hacer nada. No ocupa el procesador, sino que el proceso que ejecuta la orden se duerme durante el tiempo especificado.

```
petra:~/www$ sleep 2
20 sg. después vuelve a a aprecer el prompt
petra:~/www$
```

Esta orden incluida en un programa del shell permite retrasar la ejecución de las órdenes que vayan tras el `sleep`.

1.1.2 La orden *at*

Permite ejecutar una orden a una hora determinada. Las órdenes a ser ejecutadas deben estar contenidas en un fichero ejecutable, es decir, debe ser un programa del shell.

La orden tiene el formato siguiente:

```
at hora –f fichero
```

Para especificar la hora hay multitud de formas:

- Mediante el formato HH:MM, indicando la hora y minutos a la que se quiere ejecutar la orden. Si la hora no ha llegado se supone que es de hoy; si ya ha pasado se ejecutará a la hora indicada mañana. Podemos añadir las iniciales PM o AM para indicar la hora en formato de 12 horas, dado que de otro modo se entiende que la hora está en formato de 24 horas.
- Pueden utilizarse las palabras *midnight, noon* o *teatime*, para hacer referencia a las 12 de la noche, las 12 del mediodía o las 4 de la tarde.
- Indicando tras la hora la palabra *tomorrow* o *today* para especificar qué día queremos que se ejecute.
- Si queremos que se ejecute en otro día, tras la hora se puede indicar la fecha con el formato MM/DD/AA, siendo MM el mes, DD el día y AA el año.
- Utilizando “now + NN [minutes| hours | days| weeks]”. La orden se ejecutará dentro de NN minutos, horas, días o semanas.

```
petra:~$ at -f fichero 11:00 tomorrow
warning: commands will be executed using /bin/sh
job 36 at 2001-10-31 11:00
```

```
petra:~$ at -f fichero 11:00 today
warning: commands will be executed using /bin/sh
job 37 at 2001-10-30 11:00
petra:~$ at -f fichero 11:00 3/10/01
warning: commands will be executed using /bin/sh
job 38 at 2001-03-10 11:00
petra:~$ at -f fichero 11:00 11/3/01
warning: commands will be executed using /bin/sh
job 39 at 2001-11-03 11:00
petra:~$ at -f fichero 5:00 PM
warning: commands will be executed using /bin/sh
job 40 at 2001-10-30 17:00
petra:~$
```

Una vez introducidas las órdenes que queremos que se ejecuten podemos examinar la lista de órdenes planificadas mediante la orden `atq`:

```
petra:~$ atq
36      2001-10-31 11:00 a albizu
37      2001-10-30 11:00 a albizu
39      2001-11-03 11:00 a albizu
40      2001-10-30 17:00 a albizu
petra:~$
```

Esta orden muestra en cada caso el número de trabajo, la fecha y hora donde se va a ejecutar, tipo de orden y usuario que ha lanzado el proceso.

A la hora indicada se ejecutarán las órdenes que contenga el fichero especificado. Si la ejecución genera salida o salida de errores, éstos serán enviados por correo al usuario.

Para saber la orden que se va a ejecutar se puede utilizar la orden *at* con la opción *c*, indicando el número de trabajo. Mostrará las órdenes que se ejecutarán a esa hora, incluyendo la inicialización de todas las variables necesarias para se ejecute en el mismo entorno que ahora. La orden a ejecutar contiene lo siguiente:

```
petra:~$ cat fichero
date > hora
```

Si hemos ejecutado la orden at con este fichero como orden a ejecutar, `at -c` mostrará lo siguiente:

```
petra:~$ at -c 40
#!/bin/sh
# atrun uid=1004 gid=1000
# mail   albizu 0
umask 22
PWD=/home/profesores/albizu; export PWD
gopher_proxy=http://proxy.uniovi.es:8888/; export gopher_proxy
http_proxy=http://proxy.uniovi.es:8888/; export http_proxy
wais_proxy=http://proxy.uniovi.es:8888/; export wais_proxy
REMOTEHOST=pcpio01.ccu.uniovi.es; export REMOTEHOST
LANG=C; export LANG
HOSTTYPE=i386; export HOSTTYPE
...
OSTYPE=linux-gnu; export OSTYPE
HOME=/home/profesores/albizu; export HOME
PATH=/home/profesores/albizu/bin:.:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin
;
export PATH
cd /home/profesores/albizu || {
         echo 'Execution directory inaccessible' >&2
         exit 1
}
date > hora

petra:~$
```

Si además de especificar la hora queremos que se ejecute siempre y cuando la carga del sistema no sea excesiva (por defecto, menor que 1.5) se puede utilizar, con el mismo formato que el `at`, la orden `batch`.

Si queremos eliminar alguna orden planificada se utiliza la orden `atrm`, indicando el número de trabajo a eliminar:

```
petra:~$ atq
36      2001-10-31 11:00 a albizu
37      2001-10-30 11:00 a albizu
39      2001-11-03 11:00 a albizu
40      2001-10-30 17:00 a albizu
petra:~$ atrm 36 39
petra:~$ atq
37      2001-10-30 11:00 a albizu
40      2001-10-30 17:00 a albizu
petra:~$
```

1.1.3 La orden *crontab*

El sistema puede mantener para cada usuario una tabla con las órdenes que tiene planificadas el usuario para ser ejecutadas en momentos determinados. Para trabajar con esa tabla se utiliza la orden *crontab:*

```
crontab [-u usuario] {-e | -l | -r}
```

Con la opción `–e` permite editar la tabla del usuario; con –l muestra el contenido, mientras que la opción `–r` borra la tabla creada por el usuario.

El fichero de `crontab` contiene una línea para cada orden planificada. Cada línea tiene el siguiente formato básico:

```
Minuto hora día mes día-de-la-semana orden
```

Si en lugar de uno de los datos indicados se pone un * indica que ese dato no es significativo:

- `5 0 * * *` `orden1`: Ejecuta la orden1 a las 00:05 todos los días
- `15 14 1 * *` `orden2`: Ejecuta orden2 a las 14:15 el día 1 de cada mes.
- `0 22 * * 1-5` `orden3`: Ejecuta la orden 3 a las 22:00 de lunes a viernes
- `23 0-23/2 * * *` `orden5`: Ejecuta la orden5 a los 23 minutos de cada hora todos los días.
- `5 4 * * sun` `orden4`: Ejecuta la orden6 todos los domingos a las 4:05

En la sección 5 del manual se puede encontrar, en la página correspondiente a *crontab*, el formato completo de los ficheros de crontab.

Ejercicios

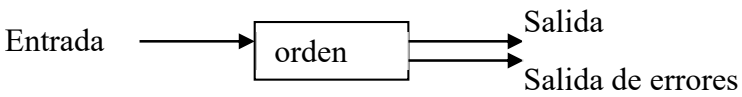
1. Localiza los usuarios que se encuentran conectados en este momento a la máquina.
2. Busca información acerca de las órdenes `who` y `finger` y comprueba la diferencia.
3. Ejecuta las líneas de órdenes:
   ```
   sleep 15; echo He terminado
   (sleep 15; echo He terminado)&
   ```
   e intenta en ambos casos ejecutar la orden `ps -f` en el tiempo de espera.
4. Ejecuta la orden `ps` para descubrir los procesos en curso.
5. Utiliza la orden `man` con `ps` para identificar las opciones de que dispone
6. Indique como podemos conocer la lista de procesos que hemos lanzado en segundo plano.
7. Indique como podemos conocer todos los procesos que el usuario tiene en ejecución.
8. Indique como podemos lanzar procesos en segundo plano.
9. Indique como podemos reanudar en segundo plano un proceso que acabamos de suspender.
10. Indique como podemos suspender el proceso que se esta ejecutando en primer plano.
11. Indique como podemos suspender un proceso que no esta en primer plano, pero que se ha lanzado desde otra terminal.
12. Indique como se eliminan procesos en ejecución. Explique las distintas alternativas que tenemos.
13. Utiliza la orden man con ps para identificar las opciones de que dispone

4. Conductos, filtros y redireccionamiento de e/s

4.1. Redirección de entrada/salida

La filosofía de funcionamiento del Unix radica en que las órdenes del shell sean sencillas de utilizar, sin tener un gran número de opciones que permitan mostrar la información de distintas maneras o seleccionar la que interesa en un momento dado. Los diseñadores de Unix decidieron en su momento hacer una clara diferencia entre las órdenes que obtienen información y las órdenes que la procesan, pudiendo manipularse la información independientemente del origen de la misma.

Por cuestiones de uniformidad, en Unix las órdenes tienen la siguiente estructura:



Las órdenes aceptan una entrada en forma de un único flujo de información y generan dos flujos de información: uno con la salida normal de la orden y otro con la salida de errores. Algunas ordenes (como todas las que se han visto hasta el momento) no aceptan ninguna información de entrada, si no que más bien generan la información a partir de datos del sistema o realizan alguna tarea concreta. Sin embargo, hay muchas órdenes que se encargan de procesar la información que se ofrece a la entrada, manipulándola y mostrándola a la salida. Esas órdenes se denominan *filtros*, pues su tarea es filtrar de alguna manera los datos de entrada mostrando por la salida únicamente los que han pasado el filtro.

Por defecto, tanto la entrada como la salida y la salida de errores están asignadas a la terminal (que se conoce como entrada o salida estándar), de tal manera que la información de entrada hay que introducirla a través del teclado y se ve la salida a través de la pantalla. En muchas ocasiones, sin embargo, podemos necesitar procesar la información almacenada en un fichero o bien almacenar la salida de una orden en un fichero para analizarla posteriormente. Unix provee un mecanismo que posibilita desviar tanto la entrada como las salidas de las órdenes hacia otro sitio distinto de la terminal. Es lo que se denomina *redirección de entrada/salida*. La forma de hacerlo a través del *shell* es por medio de los siguientes caracteres:

- Redirección de salida: `>`
- Redirección de salida de errores: `2>`
- Redirección de entrada: `<`

En el caso de la salida, si doblamos el símbolo `>` conseguiremos que no desaparezca la información almacenada el fichero especificado en caso de que ya existiera, sino que la salida de la orden se añade al final de fichero.

Así, la orden `ls -l > listado` crearía un fichero cuyo contenido es la salida de esa orden:

```
petra:~/d1$ ls
d2  f1.bis.txt  f1.txt  f3       forward       volcado.obj
f1  f1.log      f2      fichero  tmuninst.ini
petra:~/d1$ ls > listado
petra:~/d1$ cat listado
d2
f1
f1.bis.txt
f1.log
f1.txt
f2
f3
fichero
forward
listado
tmuninst.ini
volcado.obj
```

Para añadir a ese fichero la salida de la orden ps, por ejemplo, haríamos lo siguiente:

```
petra:~/d1$ ps >> listado
petra:~/d1$ cat listado
d2
f1
f1.bis.txt
f1.log
f1.txt
f2
f3
fichero
forward
listado
tmuninst.ini
volcado.obj
  PID TTY          TIME CMD
23846 pts/4    00:00:00 bash
23902 pts/4    00:00:00 ps
petra:~/d1$
```

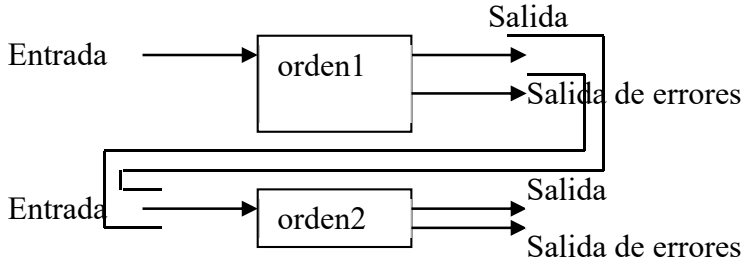
También podemos desviar la salida de errores:

```
petra:~/d1$ cd noexiste
bash: cd: noexiste: No such file or directory
petra:~/d1$ cd noexiste 2> errores
petra:~/d1$ cat errores
bash: cd: noexiste: No such file or directory
petra:~/d1$
```

Veremos ejemplos de la redirección de entrada cuando se vea algún filtro. De momento no hemos visto ninguna orden a la que se pueda aplicar esta redirección.

4.2. Conductos (Pipes)

De la misma manera que se puede desviar la salida y la entrada de una orden a un fichero determinado, se puede desviar la entrada de una orden para que obtenga los datos a tratar de la salida de otra. Sería algo similar a lo siguiente:



Es como coger una tubería para unir la salida de una orden con la entrada de la siguiente. Precisamente se utiliza esta palabra (tubería=*pipe*) para denominar esta unión de dos órdenes. En español suele traducirse por *conducto*.

Un *pipe* o conducto se especifica de la siguiente manera:

```
orden1 | orden2
```

De esta manera, el shell desvía la salida de la orden1 hacia la entrada de la orden2, ejecutando ambas órdenes en paralelo, si bien la ejecución de la segunda está condicionada por la entrega de datos por parte de la primera.

Posteriormente veremos ejemplos de pipes, una vez que se presenten algunas órdenes que admitan información de entrada.

4.3. Órdenes de visualización de ficheros

Ya conocemos una orden que permite ver el contenido de uno o de varios ficheros. Es la orden `cat`. Si visualizamos el fichero llamado `fichero`, obtenemos lo siguiente:

```
...
Linea 16 del fichero
Linea 17 del fichero
Linea 18 del fichero
Linea 19 del fichero
Linea 20 del fichero
Linea 21 del fichero
Linea 22 del fichero
Linea 23 del fichero
Linea 24 del fichero
Linea 25 del fichero
Linea 26 del fichero
Linea 27 del fichero
Linea 28 del fichero
Linea 29 del fichero

petra:~/d1$
```

Esta orden adolece de un importante problema: si queremos examinar detalladamente el contenido del fichero y éste tiene más líneas que las de la pantalla, las primeras líneas apenas las veremos, pudiendo ver con calma sólo las últimas líneas del fichero, dado que las demás “se han ido para arriba” muy rápidamente.

Para evitar ese problema podemos utilizar alguno de los paginadores de los que dispone el Unix. El primero de ellos es la orden `more`. Esta orden es un filtro que el único procesamiento de información que hace es copiar la información de la entrada estándar (o del fichero que se indique como parámetro) en la salida estándar, pero con la particularidad de que si se llena la pantalla se detiene la copia de la información hasta que el usuario pulse la barra espaciadora, momento en el cual se mostrará una nueva pantalla y así sucesivamente hasta que se complete la visualización del fichero.

```
petra:~/d1$ more <fichero
...
Linea 14 del fichero
Linea 15 del fichero
Linea 16 del fichero
Linea 17 del fichero
Linea 18 del fichero
Linea 19 del fichero
Linea 20 del fichero
Linea 21 del fichero
Linea 22 del fichero
Linea 23 del fichero
--More—
```

Un paginador más avanzado es la orden `less`. Esta orden ya la hemos utilizado sin saberlo, dado que es adonde se dirige la salida de la orden `man` una vez que ha formateado adecuadamente la información solicitada por el usuario. La orden `less` muestra el principio del fichero (tantas líneas como quepan en pantalla) y a partir de aquí acepta las órdenes del usuario para desplazarse por el fichero. Las órdenes más habituales son:

- Flecha arriba: desplaza el texto una línea hacia arriba.
- Flecha abajo: desplaza el texto una línea hacia abajo.
- Espacio o Control-f : desplaza el texto una página hacia abajo.
- Control-b : desplaza el texto una página hacia arriba.
- /*texto*: Busca el texto indicado dentro de la información mostrada por la orden.
- *n*: Repite la última búsqueda realizada, desde el punto en que nos encontramos y hacia delante.
- *b*: Repite la última búsqueda realizada, desde el punto en que nos encontramos y hacia atrás.
- *q*: Sale de la orden `less`, devolviéndonos al prompt del sistema.

Una utilidad muy común de la orden less, es utilizarla para paginar la salida de otra orden:

```
  PID TTY      STAT   TIME COMMAND
    1 ?        S      1:30 init [2]
    2 ?        SW     1:32 [kflushd]
    3 ?        SW   159:19 [kupdate]
    4 ?        SW     0:00 [kpiod]
    5 ?        SW     2:39 [kswapd]
   82 ?        S      0:00 /sbin/portmap
  147 ?        S     19:03 /sbin/syslogd
  149 ?        S      0:00 /sbin/klogd
  154 ?        S      0:00 /sbin/rpc.statd
  157 ?        SW     0:00 [lockd]
  158 ?        SW     0:00 [rpciod]
  177 ?        S      1:25 /usr/sbin/inetd
  192 ?        S      0:00 /usr/sbin/rpc.rquotad
  228 ?        S      1:59 sendmail: accepting connections on port 25
  236 ?        S      3:59 /usr/sbin/sshd
  240 ?        SL     4:41 /usr/sbin/ntpd
  244 ?        S      0:50 /usr/sbin/rpc.nfsd
  246 ?        S      0:50 /usr/sbin/rpc.mountd
  252 ?        S      0:20 proftpd (accepting connections)
  255 ?        S      0:00 /usr/sbin/atd
  258 ?        S      0:09 /usr/sbin/cron
:
```

Hemos ejecutado la orden `ps ax | less` (que no aparece en la pantalla al llenar ésta la salida de la orden). No aparece el prompt del sistema, si no que en su lugar aparece el carácter dos puntos (`:`), que es el prompt del `less`.

4.4. Filtros de uso común

Las siguientes órdenes son de uso corriente en Unix, normalmente utilizadas en combinación entre ellas para procesar la información ofrecida por alguna de las órdenes ya vistas.

- `head`: muestra las 10 primeras líneas del fichero especificado como parámetro o la entrada estándar si no se especifica ninguno. Especificando el parámetro *–n* num. muestra las num. primeras líneas de la entrada.

- `tail`: muestra las 10 últimas líneas del fichero especificado como parámetro o la entrada estándar si no se especifica ninguno. Especificando el parámetro *–n* num. muestra las num. últimas líneas de la entrada.
- `wc`: cuenta las líneas, palabras y caracteres que tiene el fichero especificado como parámetro o la entrada estándar si no se especifica ninguno. Una opción muy utilizada es *–l*, con la que sólo muestra las líneas del fichero.
- `sort`: ordena las líneas de la entrada estándar. Tiene multitud de opciones, entre las que podemos destacar:
    - -n: Ordenación numérica. Por defecto es alfabética.
    - -r: Orden inverso, de mayor a menor.
    - -u: Elimina líneas duplicadas.
    - -f: Ordena sin hacer distinción entre mayúsculas y minúsculas.
    - -b: Ignora los espacios en blanco antes de la clave de ordenación.
    - +clave: Indica la posición de las claves de ordenación. El formato de la clave se especifica siguiendo el formato P.D, donde P es la posición de la clave (comenzando por cero), mientras que D indica el desplazamiento dentro de la clave.

```
petra:~/d1$ ps x
  PID TTY      STAT   TIME COMMAND
20348 ?        S      0:00 fork1
20633 pts/13   S      0:00 -bash
22259 pts/13   R      0:00 ps x
petra:~/d1$ ps x | sort  +0.28
  PID TTY      STAT   TIME COMMAND
20633 pts/13   S      0:00 -bash
20348 ?        S      0:00 fork1
22272 pts/13   S      0:00 sort +0.28
22271 pts/13   R      0:00 ps x
```

- `uniq`: elimina líneas duplicadas. Con la opción *–c* muestra cuantas líneas repetidas había en el fichero original.
- `grep`: sirve para buscar patrones en un fichero en una lista de ficheros. El formato de la orden es el siguiente:

```
grep [opciones] expresión lista-ficheros
```

donde expresión idenfica la expresión regular a buscar dentro de la lista de ficheros. Las opciones más útiles que se pueden incluir son las siguientes:

- `-v`: Muestra las líneas que no concuerdan con la expresión.
- `-c`: Cuenta las líneas que concuerdan con la expresión.
- `-i`: Compara sin distinguir entre mayúsculas y minúsculas.

- `tr`: transforma los caracteres que se especifican presentes en la entrada por otros también indicados. El formato es el siguiente:

```
tr [opciones] car-a-cambiar [car-nuevos]
```

donde car-a-cambiar indica los caracteres que queremos traducir por los que aparecen por car-nuevos. Las opciones más habituales son las siguientes:

- `-d`: elimina todas las apariciones de los caracteres indicados en car-a-cambiar.

- `-s`: elimina los caracteres a cambiar que estén duplicados, dejando sólo uno.

```
petra:~/procesos$ who | tr [a-z] [A-Z]
NESTOR   PTS/2    NOV 13 10:44 (PCPIO50.CCU.UNIOVI.ES)
I6952468 PTS/1    NOV 13 10:10 (PCINF024.EUITIO.UNIOVI.ES)
I1632776 PTS/3    NOV 13 10:14 (PCINF038.EUITIO.UNIOVI.ES)
I2884207 PTS/4    NOV 13 10:03 (SORO166.QUIROS.UNIOVI.ES)
I1644866 PTS/10   NOV 13 09:30 (FYM06.CIENCIAS.UNIOVI.ES)
ALBIZU   PTS/13   NOV 13 09:45 (PCPIO01.CCU.UNIOVI.ES)
I1647856 PTS/16   NOV 13 10:45 (CPD38.CPD.UNIOVI.ES)
I1634644 PTS/15   NOV 13 10:42 (CM06051.TELECABLE.ES)
I1650528 PTS/17   NOV 13 10:46 (PCINF030.EUITIO.UNIOVI.ES)
I5434261 PTS/18   NOV 13 10:47 (PCINF037.EUITIO.UNIOVI.ES)
petra:~/procesos$
```

Nótese cómo se han especificado los caracteres a transformar mediante un rango de caracteres.

- `cut`: selecciona partes de cada una de las líneas del fichero de entrada. Pueden seleccionarse las partes como campos separados por un carácter determinado o como columnas dentro de cada línea. El formato de la orden es:

```
cut -flista-campos [-dsep] [-s] [lista-ficheros]
cut –clista-columnas [lista-ficheros]
```

donde:

- -*f*lista-campos: indica los campos a seleccionar. La lista puede ser un rango, un conjunto de números separados por comas o una separación de ambos.
- -*d*sep: especifica el separador entre campos (por defecto es el tabulador).
- -*c*lista-columnas: indica las columnas a seleccionar. La lista se especifica como en el caso anterior.
- -*s*: suprime el tratamiento de las líneas que no contengan el separador.

- `join`: Une líneas de dos ficheros que tengan una clave común. Ambos ficheros deben estar ordenados por la clave. El formato es el siguiente:

```
join [opciones] fich1 fich2
```

donde:

- `fich1`: especifica el primer fichero a unir. Puede indicarse como primer fichero un guion (-), en cuyo caso se tomará la entrada estándar como primer conjunto de caracteres a tratar.
- `fich2`: especifica el segundo a unir.

Las opciones más usuales son:

- `-t carácter`: Indica el separador de campo (por defecto es el tabulador).
- `-j n m`: indica la posición de los campos clave en el primer y el segundo fichero.
- `-an`: Se muestran también las líneas no concordantes de cualquiera de los dos ficheros (n puede ser 1 ó 2, para indicar fich1 o fich2).
- `lista-columnas`: indica las columnas a seleccionar. La lista se especifica como en el caso anterior.
- `-s`: suprime el tratamiento de las líneas que no contengan el separador.

- `paste`: Concatena cada una de las líneas de varios ficheros. El formato de la orden es:

```
paste [opciones] fich1 fich2
```

donde:

- `fich1`: especifica el primer fichero a unir. Puede indicarse como primer fichero un guion (-), en cuyo caso se tomará la entrada estándar como primer conjunto de caracteres a tratar.
- `fich2`: especifica el segundo a unir.

La opción más habitual es:

- `-ddelimitador`: Indica el separador de campo (por defecto es el tabulador).

4.5. Ordenes variadas

A continuación, se enumeran otras órdenes de uso menos frecuente, pero que pueden resultar útiles en alguna ocasión. Nos limitamos a indicar su funcionalidad y a mostrar un ejemplo, pudiendo obtenerse el formato detallado y el conjunto de opciones relevantes de las páginas del manual correspondiente:

– `cal`: muestra el calendario del mes en curso o del año indicado.

```
petra:~/d1$ cal
   November 2001
 S  M Tu  W Th  F  S
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

– `calendar`: muestra por pantalla los eventos para hoy y mañana recogidos en la agenda del sistema o en el fichero *calendar* o *.calendar* del directorio del usuario.

```
petra:~/d1$ calendar
Nov 14  Quarter Pounder price raised from $0.53 to $0.55 in violation of Nixon
        price controls (but okayed by Price Commission after formal request
        from McDonald's), 1971
Nov 13  Robert Louis Stevenson born, 1850
Nov 13  St. Augustine of Hippo born in Numidia, Algeria, 354
Nov 14  King Hussein's Birthday in Jordan
Nov 13  Paul Simon born, 1942
petra:~/d1$
```

– `clear`: limpia la pantalla.

– `date`: muestra la hora y la fecha del sistema.

– `tty`: muestra el nombre de terminal que tiene asociado el usuario.

– `mesg`: habilita (con el parámetro *y*) o deshabilita (con el parámetro *n*) el acceso a la terminal del usuario por parte del resto de usuarios. Si no se indica parámetro muestra si la comunicación está habilitada o inhabilitada.

```
petra:~/d1$ date
Tue Nov 13 11:39:06 CET 2001
petra:~/d1$ tty
/dev/pts/13
petra:~/d1$ mesg
is n
petra:~/d1$ mesg y
petra:~/d1$ ls -l /dev/pts/13
crw--w----    1 albizu   tty      136,  13 Nov 13 11:39 /dev/pts/13
petra:~/d1$
```

- `write`: escribe un mensaje en la terminal del usuario especificado. Para ello, tanto el emisor como el receptor deben tener habilitados los mensajes.
- `wall`: escribe un mensaje en la terminal de todos los usuarios que tengan los mensajes habilitados. Muy útil para el administrador para comunicar eventos a todos los usuarios conectados, dado que en este caso se escribe a todos los usuarios, independientemente de que tengan habilitados o no los mensajes.
- `talk`: permite establecer una comunicación interactiva con otro usuario.
- `time`: muestra el tiempo total, el tiempo en modo usuario y el tiempo en modo núcleo de ejecución de una orden concreta. Bajo este nombre hay dos órdenes: una interna al shell y otra (*/usr/bin/time*), que aporta más información que la orden interna:

```
petra:~$ time ps >/dev/null
real    0m0.344s
user    0m0.120s
sys     0m0.220s
petra:~$ /usr/bin/time ps >/dev/null
0.17user 0.11system 0:00.28elapsed 100%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (205major+134minor)pagefaults 0swaps
petra:~$
```

- `tar`: su función original es la de transferir ficheros desde/hacia un dispositivo de cinta. Suele utilizarse, sin embargo, para crear un único fichero que contenga varios, para almacenarlos en el disco, transferirlos a otra máquina, etc.

```
petra:~$ ls prueba
f1  f2  f3  f4  f5
petra:~$ tar -cf prueba.tar prueba
petra:~$ tar -tf prueba.tar
prueba/
prueba/f1
prueba/f2
prueba/f3
prueba/f4
prueba/f5
petra:~$ rm –r prueba
petra:~$ tar -xf prueba.tar
petra:~$ ls prueba
f1  f2  f3  f4  f5
petra:~$
```

- `gzip, gunzip`: Comprime /descomprime un fichero determinado, haciendo que así ocupe menos espacio.

```
petra:~$ ls -l fichero
-rwx------    1 albizu   profes       1986 Nov 13 11:51 fichero
petra:~$ gzip fichero
petra:~$ ls -l fichero*
-rwx------    1 albizu   profes        395 Nov 13 11:51 fichero.gz
petra:~$ gunzip fichero.gz
petra:~$ ls -l fichero*
-rwx------    1 albizu   profes       1986 Nov 13 11:51 fichero
petra:~$
```

Ejercicios

1. Escribe un mensaje a un compañero que esté conectado a la máquina de forma que le aparezca a éste de inmediato en pantalla.
2. Entabla un diálogo por ordenador con un compañero que esté conectado a la máquina.

3. Elimina la recepción de mensajes procedentes de otros usuarios.
4. Crea un fichero de texto con la orden `cat`.
5. Guarda el contenido de un directorio en un fichero.
6. Añade un texto a un fichero de texto ya existente.
7. Redirecciona el error que produzca una orden incorrecta a un fichero de error.
8. Cuenta el número de usuarios que están conectados al sistema en este momento.
9. Cuenta el número de usuarios que existen en el sistema.
10. Cuenta el número de grupos que existen en el sistema.
11. Cuenta el número de usuarios que existen en el sistema que tienen "García" como uno de sus dos apellidos.
12. Escribe en un fichero la información de los usuarios que tengan "Alvarez" como alguno de sus apellidos.
13. Guarda en un fichero la localización de todos aquellos ficheros del sistema que tengan como extensión .out. Realiza la operación en modo subordinado.
14. Guarda en el fichero CONECTADOS la máxima cantidad de información que puedas obtener a través de alguna orden, sobre los usuarios conectados en este momento.
15. Cuenta el número de usuarios que están conectados en este instante, mediante una orden que utilice tubería.
16. Visualiza a través de una tubería el contenido del fichero `/etc./passwd`, haciendo que la pantalla se detenga cuando ésta se llene.
17. Busca mediante una orden el fichero passwd en el directorio `/etc`.
18. Busca mediante una orden el fichero passwd en toda la jerarquía de directorios.
19. Comprueba a través de una tubería si existe algún usuario conectado en este instante que se apellide "García".
20. Crea un fichero que contenga el contenido del directorio raíz denominado RAIZ.
21. Cuente cuantos usuarios distintos hay dados de alta en petra.
22. Cuente cuantos usuarios están conectados actualmente desde una máquina de euitio.
23. Cuente cuantos usuarios están dados de alta en el sistema que se apelliden Sanchez.
24. Cuente el número de máquinas distintas desde las que se han realizado conexiones a petra este último mes (desde que se inicializo el fichero wtmp).
25. Cuente el número de usuarios distintos que se han conectado a petra este último mes (desde que se inicializo el fichero wtmp).
26. Haga una estadística (ordenada de mayor a menor) del número de sesiones que tiene abiertas cada usuario.
27. Haga una estadística (ordenada de mayor a menor) del número de veces que se está ejecutando cada proceso en el sistema.
28. Haga una estadística (ordenada de mayor a menor) del número de veces que se ha conectado a petra cada maquinas desde las que se accedido al sistema este último mes (desde que se inicializo el fichero wtmp).
29. Haga una estadística (ordenada de mayor a menor) del número de veces que se ha conectado a petra cada usuario en este último mes (desde que se inicializo el fichero wtmp).

5. Editores de texto

En el sistema operativo Unix hay un editor de texto por antonomasia, que es el *vi*. Es un editor realmente potente, pero un poco incómodo de utilizar cuando no se domina. Dada esta "incomodidad" inicial han sido muchos los editores que distintas personas e instituciones han ido realizando y, si bien no están tan difundidos como el *vi* sí es cierto que están instalados en muchos sistemas.

A continuación, se presentan los editores más habituales en un sistema Unix, si bien hay que considerar que el editor *vi* es el más difundido y el que con toda seguridad nos encontraremos en cualquier sistema, por lo que es importante que todo usuario de Unix sepa cómo funciona y tenga cierta soltura en su manejo.

5.1. El editor vi

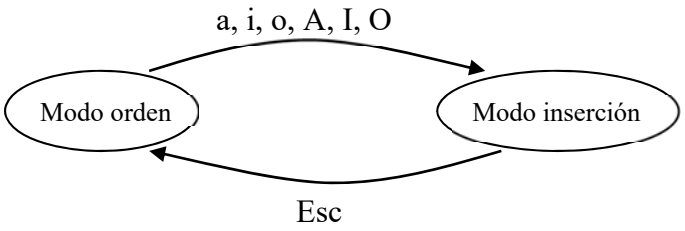
El editor *vi* es una evolución de otro editor anterior más rudimentario y que todavía se puede utilizar, que es el *ex*. De hecho, la mayor parte de las órdenes de *vi* son las de *ex*. La diferencia radica en que *ex* es un editor de líneas, mientras que *vi* es un editor de pantalla completa.

5.2. Modos de edición

El editor *vi* tiene dos modos básicos de operación: el modo orden y el modo inserción. Cuando ejecutamos *vi* comienza en modo orden, con lo cual cualquier tecla que pulsemos se va a interpretar como una orden (borrado de una línea, borrado de un carácter, copiado de una línea, etc.), pero no se ven reflejadas en la pantalla. Muchas teclas, además, no tienen asociada ninguna función, con lo que al usuario novato puede darle la impresión de que el editor no funciona.

Cuando pasamos al modo inserción por medio de la orden adecuada, que posteriormente veremos, el editor comienza a copiar lo que se va introduciendo por teclado a la pantalla. Aquí ninguna de las teclas "normales" (letras o números) no tienen ninguna función especial asociada, más que copiar el carácter pulsado a la pantalla. Sólo alguna tecla especial (Esc, algún carácter de control) van a realizar alguna función especial.

El cambio de modo se realiza según el siguiente diagrama:



Una vez que estamos en modo inserción se vuelve a modo orden pulsando la tecla *Esc*. Para pasar de modo orden a modo inserción hay 6 órdenes básicas para hacerlo. Las seis órdenes pasan a modo orden, pero se diferencian en qué posición del texto existente colocan el curso para insertar el texto:

- a: Sitúa el punto de inserción detrás del carácter actual.
- i: Sitúa el punto de inserción antes del carácter actual.
- o: Abre una línea nueva, debajo de la línea actual, colocando al principio de ésta el punto de inserción.
- A: Sitúa el punto de inserción al final de la línea actual.
- I: Sitúa el punto de inserción al principio de la línea actual.
- O: Abre una línea nueva, encima de la línea actual, colocando al principio de ésta el punto de inserción.

5.3. Órdenes de básicas de ex

Cualquiera de las órdenes del editor *ex* se puede ejecutar desde *vi*. Para ello sólo hay que introducir como orden: (el carácter dos puntos), con lo que aparece una línea de órdenes en la parte inferior de la pantalla, donde podemos introducir cualquiera de las órdenes de *ex*.

Las órdenes más habituales de este tipo que se utilizan son las siguientes:

- `write` *nombre*: Escribe el texto del editor en un fichero denominado nombre
- `read` *nombre*: Lee el fichero indicado, insertándolo tras la línea donde se encuentra el cursor.
- `quit`: Termina la sesión con el vi.
- `edit` *nombre*: Comienza la edición de otro fichero.

En todos los casos, puede abreviarse la orden con la primera letra.

En las órdenes que incluyen un nombre de fichero, en el caso que no se incluya se toma por defecto el nombre del fichero actual.

En el caso de que se haya modificado el fichero que estemos editando y ejecutemos una orden que suponga perder los cambios hechos (editar otro fichero, salir), el editor nos avisará del peligro y no realizará la orden. Si estamos seguros de que queremos hacerlo añadiremos un ! (cierre de admiración) al nombre de la orden para indicar que estamos seguros que queremos hacerlo.

5.4. Órdenes básicas de vi

Estando el vi en modo orden se pueden utilizar diversas teclas para realizar alguna acción. Hay más de 100 posibles órdenes, pero sólo mostraremos las más usadas.

Para cualquiera de las acciones de *vi* hay que tener en cuenta que, si antes de la acción introducimos un número, la acción que indiquemos ser repetirá tantas veces como el número indicado.

Movimiento del cursor

- h: mueve el cursor una posición a la izquierda.
- j: mueve el cursor una línea abajo.
- k: mueve el cursor una línea abajo.
- l: mueve el cursor una posición a la derecha.
- w: mueve el cursor una palabra a la derecha.

- b: mueve el cursor una palabra a la izquierda.
- $: mueve el cursor al final de la línea actual.
- ^ ó 0: mueve el cursor al principio de la línea actual.
- *número*G: posiciona el cursor en la línea G.

Operaciones de borrado y sustitución de caracteres

- x: borra el carácter sobre el que está el cursor.
- X: borra el carácter anterior al cursor.
- D: borra hasta el final de la línea.
- dd: borra una línea.
- r: reemplaza un carácter.
- C: reemplaza caracteres hasta el final de la línea.
- R: reemplaza caracteres (hasta que pulsemos Esc).

Operaciones varias

- /cadena: busca la cadena en el fichero desde la posición actual hacia delante.
- ?cadena: busca la cadena en el fichero desde la posición actual hacia atrás.
- n: repite la última búsqueda hacia delante.
- N: repite la última búsqueda hacia atrás.
- .: repite la última operación.
- ^L: refresca la pantalla.
- ^G: muestra el estado del fichero.
- ^Z: pasar el editor a estado suspendido.
- J: une la línea actual y la siguiente.
- Y: copia la línea actual a un buffer.
- P: inserta desde el buffer antes de la línea actual.
- p: inserta desde el buffer después de la línea actual.
- U: deshace los cambios en la línea actual.
- u: deshace la última acción.
- ZZ: salva el fichero y sale del editor.

5.5. Otros editores

Además del *vi*, y dada la dificultad inicial de manejo que presenta este editor, se han ido desarrollando por distintas personas y entidades otros editores de texto, de manejo más sencillo que el *vi* pero también con menos opciones, si bien realizan las operaciones más usuales de tratamiento de texto, con lo que son muy utilizados cuando el sistema dispone de ellos.

Casi todos disponen de ayuda en pantalla de las operaciones que pueden realizar. A continuación, enumeramos brevemente los más populares, haciendo una breve sinopsis de sus opciones más usuales.

El editor joe

Al editar un fichero con el editor *joe* se nos muestra una pantalla similar a la siguiente:

```
    IW   Unnamed                    Row 1    Col 1   11:06  Ctrl-K H for help


** Joe's Own Editor v2.8 ** Copyright (C) 1995 Joseph H. Allen **
```

Sólo mostramos la primera y la última línea de la pantalla. El resto son líneas en blanco.

Como vemos, se indica que si necesitamos ayuda se pulse Ctrl-K H. Al hacerlo, la pantalla que aparece tiene una cabecera mayor, donde se nos indica las órdenes más usuales:

```
   Help Screen    turn off with ^KH    more help with ESC . (^[.)
 CURSOR           GO TO            BLOCK      DELETE   MISC         EXIT
 ^B left ^F right ^U  prev. screen ^KB begin  ^D char. ^KJ reformat ^KX save
 ^P up   ^N down  ^V  next screen  ^KK end    ^Y line  ^T  options  ^C  abort
 ^Z previous word ^A  beg. of line ^KM move   ^W >word ^R  refresh  ^KZ shell
 ^X next word     ^E  end of line  ^KC copy   ^O word< ^@  insert   FILE
 SEARCH           ^KU top of file  ^KW file   ^J >line SPELL        ^KE edit
 ^KF find text    ^KV end of file  ^KY delete ^_ undo  ^[N word     ^KR insert
 ^L  find next    ^KL to line No.  ^K/ filter ^^ redo  ^[L file     ^KD save
    IW   Unnamed                    Row 1    Col 1    9:15  Ctrl-K H for help
```

Podemos empezar a escribir en cualquier momento. Cuando queramos realizar alguna función especial sólo tendremos que mirar la ayuda y ver cuál es la secuencia de caracteres adecuada para llevarla a cabo. Cuando tengamos soltura con las órdenes, podemos eliminar la ayuda volviendo a pulsar Ctrl-K H, con lo que nos quedará más pantalla disponible para el texto.

El editor mcedit

Al ejecutarlo vemos una pantalla como la siguiente, donde hemos recortado las líneas intermedias, dado que contendrán las líneas del fichero o estarán en blanco si el fichero es nuevo:

```
                    [----]  0:  1+ 0=  1/  1 - *0   /   0b= -1




1Help   2Save   3Mark   4Replac 5Copy   6Move   7Search 8Delete 9PullDn 10Quit
```

Vemos cómo la última línea nos indica las órdenes más habituales. En aquellas terminales que dispongan de teclas de función los números indican la tecla de función que tiene asociada la orden correspondiente. Las terminales que carezcan de estas teclas las órdenes se darán pulsando Esc y el número correspondiente a la orden (el 0 para la orden 10 Quit).

El editor pico

Otro editor sencillo de utilizar, donde la última línea nos muestra las opciones más usuales.

```
   UW PICO(tm) 2.9                New Buffer
```

```
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

Pulsando Control G se obtiene ayuda de la totalidad de las órdenes del editor. Al igual que los dos casos anteriores, podemos inmediatamente empezar a escribir sin necesidad de dar ninguna orden inicial.

Este editor es el que utiliza por defecto el programa cliente de correo *pine*.

6. Utilidades de desarrollo de programas

Existen muchas herramientas de desarrollo de programas en Unix. Aquí nos centraremos en las órdenes básicas que pueden sernos de utilidad a la hora de desarrollar programas en C.

Además, solamente ser hará mención de las herramientas desarrolladas por GNU, dado que son las que se distribuyen con los sistemas Linux. Esto no hace que sean peores que las herramientas de los sistemas Unix comerciales; más bien al contrario, las herramientas de GNU además de tener toda la funcionalidad de las herramientas equivalentes comerciales, añaden funcionalidades extra.

6.1. El compilador gcc

La herramienta básica de desarrollo de programas es el compilador. Para programas en C (el lenguaje "oficial" de Unix), este programa recibe el nombre de cc (por *C compiler*). En Linux si bien este nombre sigue existiendo no es más que un enlace al compilador real, que es el `gcc` *(GNU C compiler*). Todo lo que comentamos en este punto hace referencia a esta versión de compilador de C.

gcc no es sólo un compilador de C. En realidad, puede procesar tanto ficheros de C como de C++; en función del nombre del fichero a compilar se decidirá si es un fichero fuente en C o en C++.

En cualquiera de los casos, la tarea de `gcc` no se limita a compilar. En realidad, cuando ejecutamos la orden `gcc` se ejecutan cuatro procesos distintos:

- El preprocesador: se encarga de realizar las tareas previas a la compilación, como son la sustitución de macros o la inclusión de los ficheros que se indican en el programa.
- El compilador en sí. A partir del fichero fuente lo traduce a un programa en ensamblador.
- El ensamblador. A partir de un programa en ensamblador proveniente de la fase anterior genera los módulos objeto necesarios para su ejecución.
- En enlazador. Une distintos módulos objeto generando el ejecutable final.

El compilador de C y C++ de GNU pueden ser llamados con dos nombres distintos. Ambos llaman al mismo programa, pero varían las opciones por defecto en función del nombre de programa que se utilice:

- `gcc`: se supone que los ficheros preprocesados son de C y se utiliza el estilo de enlace de C.
- `g++`: se supone que los ficheros preprocesados son de C++ y se utiliza el estilo de enlace de C++.

En cualquier caso, el tipo de lenguaje a tratar y las acciones a realizar por defecto viene determinado por el sufijo del nombre de los ficheros fuente:

- `.c`: fichero fuente en C. Hay que preprocesarlo, compilarlo y ensamblarlo.
- `.C`: fichero fuente en C++. Hay que preprocesarlo, compilarlo y ensamblarlo.
- `.cc`: fichero fuente en C++. Hay que preprocesarlo, compilarlo y ensamblarlo.
- `.cxx`: fichero fuente en C++. Hay que preprocesarlo, compilarlo y ensamblarlo.
- `.m`: fichero fuente en Objective-C. Hay que preprocesarlo, compilarlo y ensamblarlo.
- `.i`: fichero C ya preprocesado. Hay que compilarlo y ensamblarlo.
- `.ii`: fichero C++ ya preprocesado. Hay que compilarlo y ensamblarlo.
- `.s`: fichero en ensamblador. Hay que ensamblarlo.
- `.S`: fichero en ensamblador. Hay que preprocesarlo y ensamblarlo.

En todos los casos se llevará a cabo la etapa de enlazado, a no ser que se especifique que no hay que hacerlo por medio de la opción adecuada en la llamada al compilador.

Las opciones más usuales que se indican en la llamada al compilador son las siguientes:

- `-c`: compilar y/o ensamblar, pero no enlazar.
- `-E`: Realizar únicamente la fase de preprocesamiento.
- `-S`: Parar tras la fase de compilación. Produce como salida un fichero en ensamblador.
- `-o fichero`: Indica el nombre del fichero de salida. Si no se utiliza esta opción, por defecto se generará un fichero con el nombre:
  - `a.out`, si se genera un ejecutable.
  - `nombre.o`, si lo que se genera es un fichero objeto, donde *nombre* es el nombre del fichero fuente eliminando la extensión .c o la que sea.
  - `nombre.s`, si lo que se genera es un fichero en ensamblador.
  - Si la salida debe ser el fichero preprocesado, se muestra por la salida estándar.
- `-l biblioteca`: indica la situación de la biblioteca a utilizar en la compilación.
- `-v`: Muestra por pantalla lo que va realizando en cada momento.
- `-g`: incluye en el fichero ejecutable información de depuración.
- `-ggdb`: incluye en el fichero ejecutable información de depuración específica para el depurador gdb.

Así, si creamos un programa en C y lo almacenamos en un fichero que se llame prog.c podemos compilarlo y obtener el programa en ensamblador:

```
petra:~/cc$ ls
prog.c
petra:~/cc$ gcc -S prog.c
petra:~/cc$ ls
prog.c  prog.s
```

Podemos también decidir que se compile y ensamble, pero que no se enlace para hacerlo posteriormente con otros módulos:

```
petra:~/cc$ gcc -c prog.c
petra:~/cc$ ls
```

```
prog.c  prog.o  prog.s
```

Si lo que nos interesa es realizar el proceso completo si no indicamos ninguna opción se generará un fichero de nombre a.out que contendrá el programa ejecutable correspondiente. Si quisiéramos almacenar la salida en otro fichero deberíamos introducir la opción –o:

```
petra:~/cc$ gcc prog.c
petra:~/cc$ ls
a.out*  prog.c  prog.o  prog.s
petra:~/cc$ gcc prog.c -o prog
petra:~/cc$ ls
a.out*  prog*   prog.c  prog.o  prog.s
```

Por último, si queremos realizar posteriormente la traza del programa, debemos compilarlo con la opción –g o –ggdb. Podemos ver que el tamaño del código resultante es mucho mayor que si lo compiláramos sin esta opción, dado que hay que almacenar la información necesaria para realizar la depuración.

```
petra:~/cc$ ls -l prog
-rwxr-xr-x   1 albizu   profes       24886 Nov 18 20:59 prog*
petra:~/cc$ gcc prog.c -ggdb -o prog1
petra:~/cc$ ls -l prog1
-rwxr-xr-x   1 albizu   profes      195583 Nov 18 21:00 prog1*
```

6.2. La utilidad make

La orden `make` es una utilidad que determina automáticamente qué partes de un programa han sido modificadas desde la última compilación y que, por tanto, necesitan ser recompiladas, llevando a cabo las tareas necesarias para ello.

Fundamentalmente se utiliza para desarrollar aplicaciones en C, si bien puede utilizarse con cualquier lenguaje cuyo compilador se ejecute desde la línea de órdenes. En realidad, `make` puede utilizarse para tareas no relacionadas con la compilación, sino que sirve para describir acciones que deban llevarse a cabo cuando algún fichero se modifique.

Para ejecutar el programa `make` hay que crear un fichero de nombre `makefile`, donde se especificarán las relaciones entre los distintos ficheros y las órdenes a realizar en caso de que alguno se haya modificado. Una vez creado el fichero `makefile` la orden se ejecuta simplemente indicando su nombre: `make`. Examinando las fechas que constan en los ficheros `make` decidirá que ficheros necesitan ser recompilados.

La utilidad `make` puede realizar tareas realmente complejas. En esta introducción sólo veremos su forma más sencilla de utilización.

Formato de fichero *makefile*

El fichero `makefile` consiste en un conjunto de órdenes con el siguiente aspecto:

```
objetivo : prerrequisitos
      orden
      ...
```

El objetivo es normalmente el nombre del fichero a generar por el programa, como puede ser un fichero ejecutable. También puede ser el nombre de una acción que hay que realizar.

Un prerrequisito es un fichero que se utiliza para obtener el objetivo (por ejemplo, el fichero que contiene el código fuente). Un objetivo puede depender de varios prerrequisitos.

Una orden es una acción que llevará a cabo el `make`. Puede haber varias órdenes por regla, cada una en una línea. **Cada línea de órdenes debe comenzar por un tabulador**.

Un ejemplo simple

Supóngase que tenemos un programa dividido en cinco ficheros: `practica.c`, `colas.c`, `listas.c`, `pilas.c` y `general.c`. Cada programa incluye otro fichero denominado igual que él, pero con extensión h (`practica.h`, `colas.h`, `listas.h`, `pilas.h` y `general.h`). Además, los ficheros `practica.c` y `general.c` incluyen otro denominado `defis.h`, mientras que los otros tres incluyen uno denominado `item.h`.

Cada vez que queramos obtener el ejecutable `practica` tenemos que compilar los 5 programas `.c`. El fichero `makefile` que serviría para guiar la compilación sería el siguiente:

```
practica: practica.o colas.o listas.o pilas.o general.o
	cc –o practica practica.o colas.o listas.o pilas.o \
		general.o
practica.o: practica.c practica.h defis.h
	cc –c practica.c
general.o:general.c general.h defis.h
	cc –c general.c
colas.o: colas.c colas.h item.h
	cc –c colas.c
pilas.o:pilas.c pilas.h item.h
	cc –c pilas.c
listas.o: listas.c listas.h item.h
	cc –c listas.c
borrar:
	rm practica.o colas.o listas.o pilas.o general.o
```

Véase cómo cuando se quiere dividir una línea en varias se puede hacer incluyendo al final de la línea el carácter \.

Para crear el ejecutable `practica`, lo único que hay que hacer es:

```
petra:~$ make
```

Una vez introducida la orden sólo se compilarían aquellos programas que incluyen algún fichero que ha sido modificado desde la última compilación, con el consiguiente ahorro de tiempo que suponer con respecto a una compilación completa.

Para eliminar todos los ficheros intermedios una vez concluido el desarrollo de la aplicación podríamos introducir

```
petra:~$ make borrar
```

Utilización de variables

Pueden definirse dentro del `makefile` una variable que contenga una determinada secuencia de caracteres. Una vez definida, cada vez que aparezca la variable será interpretada como si en su lugar estuviera la cadena de caracteres. Podríamos escribir ejemplo anterior de la siguiente manera:

```
ficheros = practica.o colas.o listas.o pilas.o general.o
practica: $(ficheros)
	cc –o $(ficheros)
practica.o: practica.c practica.h defis.h
	cc –c practica.c
general.o:general.c general.h defis.h
	cc –c general.c
```

```
colas.o: colas.c colas.h item.h
	cc –c colas.c
pilas.o:pilas.c pilas.h item.h
	cc –c pilas.c
listas.o: listas.c listas.h item.h
	cc –c listas.c
borrar:
	rm $(ficheros)
```

Reglas implícitas

La acción más frecuente en `make` es compilar un fichero que contiene un programa fuente para obtener el fichero objeto. Cuando necesitemos compilar un único fichero para obtener un fichero objeto del mismo nombre que el fuente (con la diferencia de la extensión) podemos evitar poner la orden en la regla correspondiente:

```
ficheros = practica.o colas.o listas.o pilas.o general.o
practica: $(ficheros)
	cc –o $(ficheros)
practica.o: practica.c practica.h defis.h
general.o:general.c general.h defis.h
colas.o: colas.c colas.h item.h
pilas.o:pilas.c pilas.h item.h
listas.o: listas.c listas.h item.h
borrar:
	rm $(ficheros)
```

El programa `make` entenderá que la orden a ejecutar para la regla, por ejemplo,

```
practica.o: practica.c practica.h defis.h
```

es `cc –c practica.c`, sin necesidad de especificarlo.

6.3. El depurador gdb

La orden `gdb` ejecuta un depurador que permite estudiar detenidamente qué sucede durante la ejecución de otro programa mientras se ejecuta, así como estudiar qué estaba haciendo un programa cuando abortó.

Fundamentalmente con `gdb` se pueden hacer cuatro cosas:

- Ejecutar el programa a depurar, especificando todo lo que pueda afectar a su comportamiento.
- Especificar puntos de ruptura, donde se detendrá la ejecución del programa, incondicionalmente o cuando se cumpla una condición determinada.
- Examinar el programa cuando éste esté parado, pudiendo consultar valores de variables, el contenido de la pila, etc.
- Cambiar cosas en el programa para experimentar la consecuencia de esos cambios.

Invocación del programa gdb

En función de lo que necesitemos depurar se puede ejecutar el depurador de tres maneras distintas:

- `gdb programa`: inicia la depuración del programa indicado.
- `gdb programa PID`: inicia la depuración del programa indicado que ya se está ejecutando bajo el proceso cuyo *PID* es el indicado.
- `gdb programa core`: examina el fichero *core* generado en una ejecución anterior fallida del *programa* indicado.

En cualquiera de las tres situaciones el programa mostrará el prompt de gdb, esperando que introduzcamos órdenes para realizar la depuración del programa:

```
petra:~/cc$ gdb prog1
GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-slackware-linux"...
(gdb)
```

Órdenes más usuales

Las siguientes son las órdenes del depurador que más habitualmente se utilizan:

- `run [parámetros]`: comenzar la ejecución del programa, pasándole los parámetros que se indican si es necesario. Se puede abreviar con **r**.
- `break [fichero:]función/línea [if condición]`: establece un punto de ruptura en la función o línea indicada (del fichero especificado si el programa está dividido en varios ficheros). Si se añade la condición sólo se detendrá la ejecución cuando se alcance el punto de ruptura y la condición sea verdadera. Se puede abreviar con **br**.
- `clear [fichero:]función/línea`: elimina el punto de ruptura especificado. Se puede abreviar con **cl**.
- `info breakpoints`: lista los puntos de ruptura definidos. Se puede abreviar con `info br`.
- `delete n`: elimina el punto de ruptura número n. Se puede abreviar con **d**.
- `continue`: continúa la ejecución del programa detenido. Se puede abreviar con **c**.
- `backtrace`: Muestra la pila del programa. Se puede abreviar con **bt**.
- `list`: lista el programa en ejecución. Se puede abreviar con **l**.
- `print expr:` muestra el valor de la expresión que se indica. Se puede abreviar con `p`.
- `display expr`: Muestra el valor de la expresión que se indica siempre que el depurador detenga su ejecución. Se puede abreviar con **d**.
- `next`: ejecuta la siguiente línea de programa; considera las llamadas a funciones como una única instrucción. Se puede abreviar con **n**.
- `step`: ejecuta la siguiente línea de programa; en caso de llamadas a funciones ejecuta una a una sus instrucciones. Se puede abreviar con **s**.
- `help`: muestra información sobre las órdenes de gdb. Se puede abreviar con **h**.
- `watch` *condición*: añade una condición de ruptura. Cuando la condición sea verdadera se detendrá la ejecución del programa. Se puede abreviar con **wa**.
- `quit`: sale de gdb. Se puede abreviar con **q**.

7. Programación del shell

El *shell* de Unix no se limita a recoger las órdenes introducidas por el usuario en la línea de órdenes. Es posible agrupar un conjunto de órdenes en un fichero y ejecutar el fichero; el shell ejecutará una a una esas órdenes.

Además, el shell permite definir variables para contener información.

En definitiva, el shell incorpora un lenguaje de programación, relativamente sencillo, pero bastante potente para automatizar las tareas más usuales de trabajo en Unix.

7.1. Modificación del entorno

Para modificar el entorno de trabajo y que estas modificaciones tengan efecto siempre que entramos en sesión, existe un fichero en el directorio de cada usuario, de nombre `.profile`[3].

Cuando se entra en sesión, tras ejecutarse el intérprete de órdenes, éste ejecuta una por una todas las órdenes contenidas en el fichero `/etc./profile`, que es común para todos los usuarios y contiene todas las órdenes que el administrador del sistema cree que debe ejecutar todo usuario para configurar su sesión de trabajo.

Dicho fichero sólo lo puede modificar el administrador del sistema, de tal manera que si un usuario quiere ejecutar una serie de órdenes cada vez que entra en sesión el sistema Unix le ofrece otra posibilidad: editar el fichero `.profile`. Dicho fichero debe estar en el directorio propio de cada usuario, y se ejecuta después de ejecutar el profile (perfil) general del sistema.

Así que cuando un usuario desea modificar su entorno, y que esas modificaciones tengan efecto siempre que se entre en sesión, lo que debe hacer es editar (o crearlo, si no existe) el fichero `.profile` de su directorio.

Las órdenes típicas que se suelen incluir suelen ser de definición de alias para las órdenes que más utiliza, modificación del camino de búsqueda para algunas órdenes, definición del tipo de terminal con el que acostumbra a trabajar, definición de teclas especiales (borrado, suspensión de procesos, ...), etc.

Variables

En el sistema se pueden definir variables, para almacenar los valores que se crean convenientes. Estos valores son siempre series de caracteres.

Para definir una variable basta con asignarle un valor. La sentencia de asignación es la siguiente:

```
variable=valor
```

---

[3] En el bash busca dentro del directorio del usuario los ficheros de nombre `.bash_profile`, `.bash_login` y `.profile`, en este orden. Ejecuta sólo el primero que exista y sea legible. Al terminar la sesión el shell ejecuta el fichero .bash_logout en el caso de que exista.

Nótese como no hay espacios en blanco entre el signo de igual y la variable y el valor. Si se introducen espacios, la orden no será reconocida. También hay que destacar que Unix es sensible a mayúsculas y minúsculas, de tal manera que la variable a no es la misma que la variable A. Es habitual nombrar a las variables siempre en mayúscula (tal vez para diferenciarlas de nombres de órdenes), pero no hay ningún inconveniente en utilizar nombres en minúsculas.

Todas las variables que tenemos definidas se almacenan en el **entorno** del shell. Podemos saber las variables que tenemos definidas utilizando la orden `set`.

Cuando queremos utilizar el valor que almacena una variable tenemos que anteponer el signo de $ al nombre de la variable. Así, la orden

```
a=b
```

asigna a la variable **a** el valor b. Sin embargo, la orden

```
a=$b
```

asigna a la variable **a** el valor que almacene la variable b.

Todas las variables que aparecen en el entorno de un determinado shell son sólo válidas para ese shell. Si queremos que esas variables se copien en el entorno de los procesos que se ejecuten desde ese shell tenemos que "exportarlas". La orden para hacer eso es `export`:

```
export a
```

Véase como lo que se exporta es la variable, no su contenido, con lo que se indica es el nombre de la variable (sin el $ delante).

Proyecto 7.1.1.1. Variables predefinidas

Hay una serie de variables que suelen estar predefinidas por el sistema. Dichas variables suelen tener una utilidad determinada, y, aunque es posible, no suele ser recomendable modificar su valor mediante asignaciones, salvo en casos concretos. Las variables predefinidas más usuales son:

- `PATH`: Indica los directorios en los que van a buscar los programas ejecutables cuando se introduce una orden tras el prompt.

- `MANPATH`: Almacena los nombres de los directorios donde se encuentran las páginas del manual en línea del sistema.

- `HOME`: Almacena el directorio propio del usuario. Cuando ejecutamos la orden *cd* sin parámetros, en realidad lo que estamos haciendo es *cd $HOME*.

-`TMOUT`: Para evitar que la gente se deje cuentas abiertas, el sistema tiene un mecanismo de tal manera que, si pasa un tiempo determinado sin entradas, la sesión se termina automáticamente. Esta variable indica ese tiempo en segundos.

- `PWD`: Indica el directorio en el que estamos situados.

- `OLDPWD`: Directorio en el que hemos estado situados antes de cambiarnos al actual.

- `SECONDS`: Número de segundos que han transcurrido desde que entramos en sesión.

- `PS1`: Aspecto del prompt principal del sistema.

- `PS2`: Aspecto del prompt secundario del sistema.

- COLUMNS, LINES: número de columnas y líneas, respectivamente, que tiene nuestra pantalla. Variables utilizadas por todos los programas de pantalla completa para saber los límites físicos de la pantalla.

Hay otra serie de variables predefinidas que se usan especialmente dentro de programas del shell. Son las siguientes:

- ?: Contiene el valor de retorno de la última orden ejecutada.

- !: Almacena el identificador del último proceso subordinado.

- $: Contiene el identificador del propio proceso.

- PPID: Es el identificador del proceso padre.

Proyecto 7.1.1.2. Entrecomillado

Muchas veces queremos asignar como valor de una variable cadenas de caracteres que incluyen blancos. Si lo hacemos directamente el sistema nos dará un error:

```
$ saludo=Buenos Dias

bash: Dias: command not found
```

Eso es debido a que el intérprete de órdenes toma el blanco como separador, con lo que encuentra dos elementos distintos: por una parte, la orden `saludo=Buenos` y por otra `Dias.`

Para poder introducir expresiones de este tipo, necesitamos entrecomillar los literales. Para una mayor potencia, Unix ofrece tres tipos distintos de entrecomillado:

- Comillas simples (' valor '): Todo lo que encierra se toma como un literal.

- Comillas dobles (" valor"): Lo que encierra se toma como un literal, a excepción de los nombres de variables (precedidas del $), que se sustituyen por su valor.

- Comillas simples inversas (` valor `): en este caso, el valor que se asigna a la variable es todo lo que dé como salida la ejecución de la orden que encierren.

Así, las asignaciones siguientes son completamente distintas:

```
a='Estamos en el directorio $PWD'

a="Estamos en el directorio $PWD"
a="Estamos en el directorio `pwd`"
```

Mientras que en el primer caso el valor que toma la variable es exactamente lo que se indica en el literal, en el segundo se sustituye la variable PWD por su valor; en el tercero se sustituye `pwd` por la salida de su ejecución.

Proyecto 7.1.1.3. Entrada / salida

La orden que nos permite imprimir por pantalla expresiones (y contenidos de variables), es la orden `echo expresion`.

La expresión es un conjunto de literales, formados cada uno por una única palabra o por más de una, en este caso convenientemente entrecomillada, según lo visto en el apartado anterior.

A la orden echo se le puede añadir la opción `-n` para evitar que imprima un retorno de carro al final de las expresiones que se han introducido.

Si, por contra, queremos incluir retornos de carro dentro de una expresión, podemos hacerlo mediante la secuencia de escape `\n`. Si lo que queremos es introducir tabuladores, lo haremos mediante la secuencia `\t`. Hay bastantes más caracteres de escape que podemos introducir, pero son de rara utilidad.

De la misma manera que instrucciones de salida, también las hay de entrada. La orden de entrada es `read variable`. En este caso se asignará a la variable el valor que se introduzca por teclado.

Proyecto 7.1.1.4. Operaciones con variables

Hay una serie de operaciones que están definidas sobre variables. Son las siguientes:

- `${variable}expresión`

Concatena al valor de la variable la expresión.

- `${variable:-valor}`

Si la variable no está definida o tiene un valor nulo, utiliza el valor dado.

- `${variable:+valor}`

Si la variable está definida y tiene un valor distinto de nulo, utiliza el valor dado.

- `${variable:=valor}`

Si la variable no está definida o tiene un valor nulo, utiliza el valor dado y se lo asigna a la variable.

- `${variable:?mensaje}`

Si la variable no está definida o tiene un valor nulo, imprime el mensaje indicado. En el caso que se omita el mensaje se da un mensaje por defecto (`parameter null or not set`).

- `${#variable}`

Devuelve la longitud (número de caracteres) del contenido de la variable.

- `${variable%expresion}`

Elimina el lado derecho de la cadena, siempre y cuando el izquierdo se ajuste a la expresión indicada. Dicha expresión es una expresión regular, compuesta con los caracteres comodín conocidos.

- `${variable#expresion}`

Elimina la parte izquierda del valor de la variable, siempre y cuando la parte derecha se ajuste a la expresión.

Ejemplos:

```
$ NOMBRE=fichero.extension
$ echo ${NOMBRE%.*}
fichero
$ echo ${NOMBRE#*.}
extension
```

7.2. Ficheros de órdenes (Programas del Shell)

Los programas del shell son, simplemente, fichero de textos que contienen una serie de órdenes de Unix, que ya las conocemos; de manera que cuando se ejecuta un programa, se ejecutan una a una las ordenes que contiene, de la misma manera que si hubiéramos

introducido una por una las ordenes por el teclado. Podemos utilizar todas las órdenes que ya sabemos, variables, operaciones con variables, ..., además de una serie de estructuras de control que iremos presentando posteriormente.

Ejecución de programas del shell

Hay varias formas de ejecutar ficheros de órdenes:

- Modificando los permisos para hacer ejecutable el fichero. En este caso para ejecutarlo bastaría con introducir el nombre del fichero directamente tras el prompt, una vez que el fichero tenga permiso de ejecución para el usuario que quiera ejecutarlo. El realizar la ejecución de esta manera conlleva la ejecución de un nuevo shell, que tomará como entrada (órdenes a ejecutar), las que aparezcan en el fichero de órdenes.

Teniendo en cuenta esta manera de ejecución, todos los cambios de entorno que se realicen en el programa no afectarán al shell principal. Además, las variables de entorno a las que podrá acceder el programa serán, además de las que defina él mismo, las que se habían *exportado* en el shell principal.

Ejemplo:

Sea un programa del shell almacenado en el fichero ejemplo1, con permisos para ejecución, que contenga las siguientes órdenes:

```
echo $A
A=Adios
```

y ejecutamos la siguiente secuencia de órdenes:

```
$ A=hola
$ ejemplo1

$ echo $a
hola
```

donde la única salida que se obtiene es hola al no estar definida la variable A en el subshell. Sin embargo, si ejecutamos

```
$ export A=hola
$ ejemplo1
hola
$ echo $a
hola
```

la variable A del programa tendrá valor porque se ha copiado el del shell principal, al haber "exportado" la variable.

Nótese como, en ambos casos, el programa no cambia el valor de la variable A del shell principal.

- Ejecutando un nuevo shell, pasándole como parámetro el fichero de órdenes. En este caso la ejecución se realizaría mediante la orden:

```
bash fichero
```

Todo lo que ocurre es idéntico al caso anterior.

- Ejecutando el fichero de órdenes en el propio shell. En este caso, no se crearía un shell distinto, sino que se irían ejecutando una a una todas las órdenes del programa, utilizando, por tanto, el entorno del propio shell. El ejemplo anterior, en este caso sería:

```
$ A=hola
$ . ejemplo1
hola
$ echo $a
adios
```

Véase cómo la forma de indicar que vamos a ejecutar un programa dentro del propio shell es poniendo un punto antes del nombre del fichero de órdenes. Tanto en este caso como en el anterior no es necesario que el fichero tenga permiso de ejecución.

A veces nos interesa ejecutar de la misma manera un conjunto de órdenes, pero sin editar un programa. Podemos hacer esto agrupando entre paréntesis las órdenes que queremos que se ejecuten. Por ejemplo:

```
$ ( cd directorio; pwd; ps)
```

Esto crearía un shell nuevo para ejecutar las órdenes, con lo que sería aplicable todo lo explicado anteriormente para este caso.

Si agrupamos las órdenes entre llaves se ejecuta en el mismo shell.

```
$ { cd directorio; pwd; ps }
```

con el resultado antes comentado.

La ejecución de los dos ejemplos anteriores sería la siguiente:

```
petra:~$ ( cd directorio; pwd; ps)
/home/otros/zubi/directorio
  PID TTY STAT  TIME COMMAND
22502 pp2 S     0:00 -bash
22890 pp2 S     0:00 -bash
22891 pp2 R     0:00 ps

petra:~$ { cd directorio; pwd; ps }
/home/otros/zubi/directorio
  PID TTY STAT  TIME COMMAND
22502 pp2 S     0:00 -bash
22924 pp2 R     0:00 ps
petra:~/directorio$
```

Como se puede observar, en el primer caso se están ejecutando dos intérpretes de órdenes (bash), mientras que en el segundo sólo hay uno. Además, en el segundo caso el directorio en el que quedamos tras ejecutar la orden ha cambiado, mientras que en el primer caso no.

Paso de parámetros

Suele resultar muy útil la posibilidad de introducir parámetros en la línea de órdenes. Esto nos posibilita realizar programas que hagan algo en cada ejecución con los parámetros que nos interesen en cada caso.

Cuando ejecutamos un programa del shell, éste copia los valores introducidos tras el nombre de la orden en unas variables especiales: `$1`, `$2`, ..., `$9`. Estas variables suelen llamarse *parámetros posicionales*.

Así, si introducimos en la línea de órdenes:

```
$ programa fich1 fich2 uno
```

dentro del fichero de órdenes programa la variable `$1` tendría el valor `fich1`, la variable `$2` el valor fich2 y la variable `$3` el valor uno.

En relación a los parámetros posicionales, además de esas variables disponemos de otras dos: `$#` indica el número de parámetros que se han introducido; `$*` contiene todos los parámetros.

Muchas veces nos va a interesar introducir parámetros utilizando caracteres comodines, al igual que hacemos en el resto de las órdenes de Unix. No hay ningún problema: el shell se encarga de interpretar esas expresiones regulares y de pasar al programa la lista de los nombres de ficheros que cuadran con esa expresión.

Como hemos indicado, solamente son accesibles directamente los nueve primeros parámetros. Esto no significa que no podamos pasar más parámetros a los programas del shell. Podemos acceder al resto de los parámetros por medio de la orden `shift`. Esta orden desplaza los parámetros posicionales, asignando a `$1` el valor de `$2`, a `$2` el de `$3`, ..., y a `$9` el valor que tuviera el décimo parámetro. El valor que tenía el primer parámetro se pierde, y el valor del `$#` se decrementa en uno.

Comentarios

Siempre suele ser interesante la posibilidad de introducir comentarios dentro de los programas que se hacen. En programas del shell es posible mediante el carácter `#`: todo lo que siga a este carácter hasta el fin de línea será considerado como un comentario.

Documentos *here*

Es otra forma de efectuar redirecciones de entrada. Sirven para redirigir la entrada de una orden a una serie de caracteres que están escritos a continuación. Por ejemplo, podríamos poner en un programa:

```
mail usuario <<fin
Esto es una prueba de correo.
Podemos escribir todo lo que queramos.
La entrada terminara con la palabra magica
fin
```

Todo lo que hay escrito hasta que aparece la palabra que sigue al símbolo de redirección (<<) será llevado a la entrada estándar de la orden de que se trata.

7.3. Estructuras de control

En un programa del shell se pueden introducir todas las órdenes que hemos visto hasta ahora, pero hay además otra serie de órdenes que pueden ser interesantes, que nos van a permitir realizar bucles, estructuras condicionales, etc.

Todas estas órdenes se pueden introducir directamente tras el prompt del sistema, sin necesidad de realizar un fichero de órdenes, pero es en éstos donde muestran su verdadera utilidad.

Ejecución condicional simple

La primera estructura condicional disponible en el shell de Unix es la siguiente:

```
if orden
then
  ordenes
else
 ordenes
```

```
fi
```

Si la orden que aparece tras el *if* se ejecuta con éxito (es decir, si devuelve un código de retorno igual a cero), se ejecutan las órdenes que hay tras el *else*. Si no es así, se ejecutan las órdenes que hay tras el *else*. Esta segunda parte es opcional, pudiendo utilizar únicamente la estructura *if-then-fi.*

Una estructura similar permite anidar instrucciones condicionales de una manera sencilla. En la construcción anterior nada impide que cualquiera de las órdenes que ahí se indican sean instrucciones de condicionales, pero a veces puede resultar más fácil utilizar una instrucción como la siguiente:

```
if orden1
then
 ordenes2
 elif orden3
     ordenes4
 else
     ordenes5
  fi
```

En este caso, si la *orden1* acaba con éxito se ejecutan las *ordenes2*; si no es así, se ejecuta la *orden3*, y si ésta tiene éxito, se ejecutan las *ordenes4*, mientras que si acaba en fracaso, se ejecutarían las *ordenes5.*

Esta estructura se puede generalizar, pudiendo haber tantos *elif* como se deseen, pero sólo es posible tener un *else*.

El problema que aparece con este tipo de construcción es que todo el mundo está habituado a realizar acciones condicionales en función del valor de una expresión lógica, no en función de cómo se haya ejecutado una determinada orden. Para poder seguir pensando de la misma manera el shell de Unix dispone de la orden **test**:

```
test expresion
```

devuelve un valor igual a cero (éxito) si la expresión es verdadera; si no es así devuelve un valor distinto de cero (fracaso).

Una notación alternativa, que puede quedar más clara, es la siguiente:

```
[ expresion ]
```

La expresión podemos construirla con los siguientes operadores:

Operaciones con enteros

(n1 y n2 contienen enteros)

n1 -eq n2	Cierto si n1 y n2 son iguales (equal)
n1 -ne n2	Cierto si n1 y n2 son distintos (not equal)
n1 -gt n2	Cierto si n1 es mayor que n2 (greater than)
n1 -ge n2	Cierto si n1 es mayor o igual que n2 (greater or equal)
n1 -lt n2	Cierto si n1 es menor o igual que n2 (lower than)
n1 -le n2	Cierto si n1 es menor o igual que n2 (lower or equal)

Operaciones con cadenas de caracteres

(s1 y s2 contienen *strings*)

-z s1	Verdadero si la longitud del contenido de s1 es cero
-n s1	Verdadero si la longitud del contenido de s1 no es cero

s1 = s2	Verdadero si s1 y s2 son iguales
s1 != s2	Verdadero si s1 y s2 no son iguales
s1	Verdadero si s1 no es la cadena nula

Operaciones con ficheros

(fich es el nombre de un fichero)

-a fich	Cierto si fich existe
-r fich	Cierto si podemos leer el fichero fich
-w fich	Cierto si podemos escribir el fichero indicado
-x fich	Cierto si podemos ejecutar el fichero indicado
-f fich	Cierto si fich existe y es un fichero normal
-d fich	Cierto si fich existe y es un directorio
-h fich	Cierto si fich existe y es un enlace simbólico
-c fich	Cierto si fich existe y es un fichero de dispositivo de caracteres
-b fich	Cierto si fich existe y es un fichero de dispositivo de bloques
-s fich	CIerto si s existe y tiene un tamaño mayor que cero

Operaciones con expresiones

(exp1 y exp2 son expresiones formadas con los operadores anteriores o éstos mismos)

! exp1	Verdadero si exp1 es falso
exp1 -a exp2	Verdadero si exp1 y (*and lógico*) exp2 son verdaderas
exp1 -o exp2	Verdadero si exp2 o (*or lógico*) son verdaderas

Instrucción case

La instrucción case es útil cuando queremos comparar el valor de una variable con una serie de valores distintos, realizando acciones distintas en cada caso. La estructura de esta orden es la siguiente:

```
case cadena
in
lista1)
     ordenes1
     ;;
lista2)
     ordenes2
     ;;

....
esac
```

Si el valor de la cadena (que puede ser perfectamente el contenido de una variable) está en la lista1 se ejecutan el conjunto de órdenes 1 y pasa a la siguiente orden al *esac*; si está en la segunda lista se ejecuta el segundo conjunto de órdenes, y así sucesivamente. Los elementos de cada lista deben estar separados por el símbolo |.

Las listas pueden estar constituidas por expresiones regulares, de tal manera que si el valor de la cadena cuadra con la expresión regular se ejecutarían las órdenes correspondientes.

La posibilidad de introducir expresiones regulares puede ser interesante para realizar acciones por defecto. Por ejemplo, el siguiente programa

```
read V
```

```
case $V in
     [Ss]*)
          echo  "Contestación afirmativa"
          ;;
     [Nn]*)
          echo "Constestación negativa"
          ;;
     *)
          echo "Contestación no definida"
          ;;
esac
```

tiene una opción por defecto. En el caso que se introduzca un valor de la variable que no comience por s o por n (mayúsculas o minúsculas), ejecutará las órdenes que hay tras el *, dado que con esta expresión regular cuadran todos los valores.

El bucle for

Es la primera posibilidad que tenemos para realizar iteraciones. Para cada valor de los que indiquemos en una lista se realiza una iteración:

```
for i in lista
do
  ordenes
done
```

En este bucle, la variable de control *i* irá tomando sucesivamente cada uno de los valores contenidos en la lista, y para cada uno de esos valores se ejecutarán las órdenes que se indican.

La parte donde se indica la lista de valores (*in lista*) se puede suprimir. En este caso la lista de valores que toma el bucle es el conjunto de parámetros que se pasa al programa. Los dos programas siguientes serían equivalentes:

```
for i in $*
do
  ordenes
done
```

```
for i
do
  ordenes
done
```

También se puede especificar como lista una expresión regular. En este caso la lista de valores se forma por los nombres de ficheros que se ajusten a esa expresión regular.

La última posibilidad que tenemos es especificar la lista como cada uno de los elementos de la salida de una orden determinada. Para ello podemos utilizar el entrecomillado que conocemos:

```
for i in `who | cut -f1 -d" "`
do
  write $i <<fin
     Hola a todos
  fin
done
```

Mandaría el mensaje "Hola a todos" a todo el mundo que se encuentra conectado al sistema en este momento; sería similar a la orden **wall**.

Proyecto 7.3.1.1. Bucles con un número de iteraciones conocido

Un problema que presenta el bucle *for* del shell respecto a bucles de otros lenguajes de programación es que realiza una iteración para cada elemento de una lista, mientras que en otros lenguajes puedes especificar que realice una iteración para cada elemento desde el 1 hasta el 20, por ejemplo.

Sin embargo, podemos hacer que el *for* del shell se comporte de manera parecida al de otros lenguajes. Para ello podemos utilizar la orden ***seq***:

```
albizu@petra:~$ seq 1 5
1
2
3
4
5
```

Esta orden genera una lista de números desde el indicado como primer parámetro hasta el señalado como último. Eso lo podemos utilizar para hacer bucles con un número de iteraciones conocido:

```
for i in `seq 1 10`
     do
      echo "Es la interacion $i"
     done
```

Mostraría 10 líneas, cada una de ellas con el mensaje "Es la iteración 1" (ó 2, ó 3, etc.)

Incluso si queremos que el índice del bucle no se incremente de 1 en uno, sino que lo haga con otro paso, podemos utilizar la misma orden *seq*, introduciendo un tercer parámetro, que es el número de paso. Hay que tener cuidado en este caso porque el paso lo indica el **segundo** parámetro (y no el tercero, como podría parecer más lógico). Por ejemplo, el siguiente código mostraría los números impares entre 1 y 100:

```
for i in `seq 1 2 100`
     do
      echo –n "$i "
     done
```

Los bucles while y until

Cuando queremos realizar un conjunto de iteraciones hasta que se cumpla una condición (o hasta que deje de cumplirse) podemos utilizar las estructuras while o until:

```
while orden          until orden
do                   do
 ordenes               ordenes
done                 done
```

El primer bucle (*while*) se ejecuta mientras que la orden se ejecute con éxito; el bucle *until* se ejecuta mientras la orden se ejecute con fracaso.

Una vez más, la orden más habitual que controla la ejecución del bucle suele ser la orden **test**, que nos permite en este caso realizar la iteración en función del valor de una condición.

Proyecto 7.3.1.2. Ruptura de control en un bucle: órdenes break y continue

En algún momento puede resultar útil el no realizar todas las iteraciones que estaban previstas en el bucle (dentro de un bucle *for*), o bien antes de que se cumpla (o se deje de cumplir) la condición de salida (bucles *while* y *until*).

La orden **break** termina la ejecución del bucle en el que está más inmediatamente incluido, pasando a ejecutar la orden siguiente al *done* correspondiente. Si le ponemos un parámetro numérico n al **break** saldrá del n-ésimo bucle (contando de dentro hacia afuera) anidado que lo contenga.

La orden **continue** hace que la ejecución pase a la primera instrucción del bucle, comenzando la iteración siguiente. De la misma manera que en el *break*, se puede acompañar un valor numérico para ejecutar la siguiente iteración de un bucle anidado más externo.

Proyecto 7.3.1.3. Bucles indefinidos. Órdenes true y false

Hay dos órdenes cuya principal finalidad es realizar bucles indefinidos: las **true** y **false**. Ambas órdenes no hacen nada más que devolver un valor de retorno. La orden **true** devuelve siempre éxito, mientras que la orden **false** devuelve siempre fracaso.

Siempre que se utilicen estas órdenes con un bucle *while* o *until* entre las instrucciones que haya dentro del bucle debería encontrarse alguna orden *break*, para poder terminar la ejecución normalmente.

7.4. Realización de menús: la orden select

Es habitual querer realizar programas que consten de un menú, donde se muestran las distintas opciones que el programa puede ejecutar, debiendo el usuario escoger una. Una vez escogida la opción suele volver a presentarse el menú, para que se escoja otra opción.

El realizar un programa de este tipo con las instrucciones vistas hasta ahora no sería ningún problema, pero el shell de Unix ofrece una posibilidad adicional, por medio de la orden **select**.

En realidad, la orden **select** es una orden de iteración: implementa un bucle indefinido, pero con la particularidad que la propia orden imprime el menú que le indicamos y lee la opción que el usuario introduzca. Su sintaxis es:

```
select var in lista
do
  ordenes
done
```

donde *lista* es el conjunto de opciones del menú.

El funcionamiento de la orden es el siguiente: primero imprime cada uno de los elementos que aparecen en la lista (que no es más que un conjunto de elementos separados por blancos), anteponiendo a cada uno un valor numérico. A continuación imprime la variable PS3 y espera a que el usuario introduzca un número de los mostrados (que el usuario elija una opción). El elemento de la lista que acompañe a ese número será asignado a la variable *var*, mientras que la variable REPLY contendrá el número introducido.

Por ejemplo, la ejecución del programa

```
$PS3="Introduce tu opción"
select a in "Opcion primera" "Opcion segunda" "Opcion
tercera"
do
 echo "Has introducido un $REPLY que corresponde con
$a"
 echo
done
```

produciría la siguiente salida:

```
1) Opcion primera
2) Opcion segunda
3) Opcion tercera
Introduce tu opcion: 1
Has introducido un 1 que corresponde con Opcion
primera

1) Opcion primera
2) Opcion segunda
3) Opcion tercera
Introduce tu opcion: 3
Has introducido un 3 que corresponde con Opcion
tercera

1) Opcion primera
2) Opcion segunda
3) Opcion tercera
Introduce tu opcion:
```

La ejecución del bucle no terminaría hasta que se ejecute una orden *break* o se introduzca como contestación el carácter fin de fichero.

Lógicamente, si queremos realizar un menú en condiciones, tras la orden *select* debería realizarse una ejecución condicional (mediante *case* o *if*), que en función del valor introducido por el usuario ejecute una cosa u otra.

7.5. Opciones en la línea de órdenes. La orden getops

Muchas veces que se realiza un programa del shell suele hacerse de manera que, además de poder pasarle parámetros, podamos introducir opciones que modifiquen en algo el comportamiento del programa, de manera similar a como hacen las órdenes normales de Unix (por ejemplo, *ls -ld* modifica el comportamiento de la orden *ls*).

Todas las órdenes de Unix siguen una serie de normas de sintaxis, para que la forma de trabajar con todas las órdenes sea lo más uniforme posible. De entre esas reglas podemos destacar las siguientes:

- El formato genérico de una orden es:

```
orden [opciones] [parámetros]
```

donde tanto las opciones como los parámetros pueden ser opcionales u obligatorios según la orden de que se trate.

- En el caso que puede haber más de un parámetro, esto se especifican uno tras otro, separados por uno o más espacios en blanco.

- Las opciones van precedidas por un guion (-).

- El orden de las opciones es indiferente.

- Los nombres de las opciones están constituidos por una única letra.

- Las opciones pueden requerir un parámetro.

- Las opciones sin argumentos pueden agruparse en cualquier orden.

Por tanto, si queremos realizar un programa bien hecho, deberemos seguir estas normas.

Para facilitar el análisis de opciones y comprobar su validez, el shell dispone de la orden **getopts**. Su formato básico es el siguiente:

```
getopts cadena variable
```

La *cadena* contiene los caracteres de opción que deben ser reconocidas. Las opciones que necesiten un parámetro adicional van seguidas de dos puntos (:).

Cada vez que se llame a la orden **getopts** ésta asignará el valor de la siguiente opción que se haya introducido en la línea de órdenes (siguiendo las reglas antes descritas) a la *variable* indicada, además de asignar el índice de la siguiente opción a procesar en la variable OPTIND, y el valor del parámetro que acompaña a la opción (en su caso) a la variable OPTARG.

Por ejemplo, supóngase que el siguiente programa está almacenado en el fichero ejemplo:

```
while getopts abc: n
do
  echo "$n $OPTIND echo $OPTARG"
done
```

Obtendríamos las siguientes salidas para las distintas ejecuciones:

```
$ ejemplo -ba -c fichero
b 1
a 2
c 4 fichero
```

Donde en la primera iteración la variable n tomaría el valor de la primera opción introducida (b), OPTIND valdría 1, puesto que la siguiente opción a leer está todavía en el parámetro posicional 1, mientras que OPTARG sería nulo al no llevar el signo: tras la b en la orden *getopts.*

En la segunda iteración los valores reflejados son la segunda opción introducida, el 2 puesto que la siguiente opción está en el segundo parámetro posicional, mientras que no hay argumento opcional.

En la tercera iteración sí hay argumento opcional (*fichero*), con lo que sí tendría valor la variable OPTARG.

7.6. Funciones en programas del shell

Otra característica útil en programación en general es la posibilidad de definir funciones, como conjuntos de órdenes que se van a ejecutar en distintas partes del programa, con lo que es preferible tener un mecanismo para escribir las órdenes solamente una vez y disponer de algún mecanismo que nos permita indicar que queremos ejecutar ese conjunto de órdenes.

El shell de Unix también dispone de esta posibilidad. Cuando se ejecuta una función las instrucciones se ejecutan en el entorno del propio shell; es decir, no se crea un nuevo shell. La forma de definir funciones es la siguiente:

```
function nombre ()
{
  ordenes
}
```

Para llamar a la función basta con poner su nombre.

Es posible pasar parámetros a la función, simplemente indicándolos en la llamada tras el nombre de la función.

Dentro de la función podemos acceder a los parámetros que se han pasado de la misma manera que en el resto del programa accedemos a los parámetros posicionales, utilizando las variables $1, ..., $9, $#, la función *shift*, etc.

7.7. Evaluación de expresiones aritméticas

El shell de Unix no está pensado para realizar programas de cálculo numérico. Por ello, no dispone de la potencia de evaluación de expresiones matemáticas que puedan tener otros lenguajes. A pesar de ello, dispone de un conjunto de órdenes, cuyo funcionamiento es muy similar, para realizar pequeñas operaciones aritméticas que pueden hacer falta en algunos programas.

La primera de estas órdenes es **expr**. Su formato es

```
expr expresion
```

Esta orden toma los parámetros que se le pasan como elementos de una expresión aritmética (cada elemento debe estar separado de los demás por uno o más blancos). La salida de la orden es el resultado de evaluar la orden.

Los operadores que se pueden utilizar en la orden son los siguientes:

+	Adición	\&	AND lógico
-	Sustacción	|	OR lógico
*	Multiplicación	:	Comparación de exp. regulares
/	División		

Nótese cómo cuando utilizamos un carácter que pueda ser interpretado por el shell de otra manera (como el *, & o |) debemos introducir antes el carácter de escape (\).

Las operaciones lógicas funcionan de la siguiente manera:

```
expr expresion1 \| expresion2
```

Si la *expresion1* no es nulo o cero, devuelve *expresion1*; si no, devuelve *expresion2*.

```
expr expresion1 |& expresion2
```

Devuelve expresion1 si tanto expresion1 como expresion2 son no nulos o cero. Si no, devuelve 0.

```
expr expresion1 : expresion2
```

En este caso, ambas son expresiones regulares. Devolverá cero si ninguna de las dos expresiones coincide; si no, devuelve el número de caracteres coincidentes.

Si queremos asignar a una variable el valor de una expresión aritmética, tendremos que utilizar el entrecomillado necesario:

```
a=`expr $a + 1`
```

Esta forma de trabajar puede resultar bastante incómoda. Existe una orden más útil cuando queremos realizar operaciones:

```
let var=expresión
```

Esta orden asigna a la variable *var* el resultado de la evaluación de la expresión. Los operadores son los conocidos +, -, *.

7.8. Depuración de programas

El shell también una opción para depurar programas. Cuando tengamos que hacerlo, basta con ejecutar el programa ejecutándolo en un bash llamado con la opción x:

```
bash -x programa
```

Esto hará que el shell imprima cada una de las órdenes que va ejecutando antes de ejecutarla, con lo que podemos observar por donde se va desarrollando la ejecución. Por ejemplo, si ejecutamos el siguiente programa con la opción de depuración:

```
while [ $# -gt 0 ]
do
 echo $1
 shift
done
```

obtendríamos el siguiente resultado:

```
$ bash -x ejemplo dos tres
+ [ 2 -gt 0 ]
+ echo dos
dos
+ shift
+ [ 1 -gt 0 ]
+ echo tres
tres
+ shift
[0 -gt 0 ]
```

donde podemos ir observando qué va ejecutando el shell en cada momento. Las líneas que comienzan por un + son las órdenes que se van ejecutando, mientras que las que no tienen ese símbolo son la salida del programa. Nótese cómo sustituye el contenido de las variables, para que podamos ver qué valores van tomando.

7.9. Terminación del programa

Si en un momento determinado queremos terminar el programa, basta con ejecutar la orden **exit**. Esta orden terminará con el shell que se está ejecutando, devolviendo el control a su padre. Nótese cómo si utilizamos esta orden en un programa que ejecutemos con la orden **.** (ejecución en el propio shell), acabaría la sesión en la que estamos trabajando.

Como hemos visto anteriormente, cuando un programa termina devuelve a su padre un código de retorno. Podemos indicar tras el *exit* el código que queremos devolver. Por convenio se devuelve 0 si el programa finaliza sin error y otro valor para indicar errores. En caso que no introduzcamos un valor tras el *exit* se devolverá el valor devuelto por la orden ejecutada inmediatamente antes que el *exit*.

Ejercicios

1. Realice un programa del shell que muestre por pantalla los números del 1 al 100
2. Realice un programa del shell que pase a minúsculas los nombres de todos los ficheros del directorio actual.
3. Realice un programa del shell que edite un fichero (con cualquiera de los editores disponibles) en caso de que tengamos permiso de lectura y escritura sobre él. En

caso de que sólo tengamos permiso de lectura, que lo visualice con el *less* y si no tenemos ninguno de los dos que nos muestre un error por pantalla.

4. Realice un programa del shell que cambie el directorio actual al directorio raíz del sistema. ¿Cómo debo ejecutarlo?
5. Realice un programa del shell que renombre todos los ficheros que hay en el directorio actual añadiendo a su nombre la extensión ".viejos".
6. Realice un programa del shell que renombre todos los ficheros que se le pasan como parámetro añadiendo a su nombre la extensión ".viejos".
7. Realice un programa del shell que, a partir de un fichero de texto, cree tantos ficheros como líneas tenga, conteniendo cada uno de ellos una línea del fichero. El nombre de cada fichero será el mismo que el fichero original, pero añadiendo un . y el número de la línea de contiene.
8. Realice un programa del shell que "borre" ficheros. El programa lo que hará será mover los ficheros que se le pasen como parámetros a un directorio denominado ".papelera" dependiente del directorio $HOME del usuario. En caso de que no exista el directorio, deberá crearse. En caso de que ya exista un fichero con ese nombre en la papelera, se le añadirá un . y un número al final de su nombre (si borro el fichero f y ya existe uno con este nombre, lo intentaré con f.1; si ya existe uno con este nombre, lo intentaré con f.2; así sucesivamente hasta que encuentre uno que no exista).
9. Realice un programa del shell que pare todos los procesos que tiene el usuario, a excepción del propio shell que está ejecutando el programa.
10. Realice un programa del shell que elimine todos los procesos que tiene el usuario, a excepción del propio shell que está ejecutando el programa.
11. Realice un programa del shell que mande una señal que pasaremos como parámetro a todos los procesos que tiene el usuario, a excepción del propio shell que está ejecutando el programa.
12. Realice un programa del shell que muestre para cada usuario el nombre del login, el nombre real de usuario y los nombres de los grupos a los que pertenece.
13. Realice un programa del shell que cree 50 ficheros en el directorio, que se llamen fich1, fich2, ..., fich50 y que el contenido de cada fichero sea 5 líneas con el propio nombre del fichero.
14. Realice un programa del shell que muestre, para cada uno de los usuarios que están conectados en el sistema, su directorio HOME.
15. Realice un programa del shell que renombre todos los ficheros del directorio que se apelliden .C, cambiando sus nombres a .c.
16. Realice un programa del shell que renombre todos los ficheros del directorio, pasando sus nombres a minúsculas.

Apéndice 1: Resumen de mandatos del bash

A continuación se muestra la relación de todas las órdenes internas del bash. Muchas ya han sido estudiadas con anterioridad. En todos los casos se ofrece una descripción muy somera de la funcionalidad de la orden, pudiéndose obtener la información completa utilizando la orden *help* dentro del sistema.

`%[DIGITS | WORD] [&]`	Reanuda la ejecución de un proceso parado con anterioridad. La reanudación se efectuará en segundo plano si se incluye el &.
`. filename`	Ejecuta el *shell script* contenido en el fichero indicado dentro del propio shell.
`:`	No hace nada. Es una orden nula.
`[ arg... ]`	Evalúa condiciones (similar al *test*).
`{ COMMANDS ; }`	Ejecuta una lista de ordenes en el propio shell.
`alias [-p] [name[=value] ... ]`	Define nombres alternativos para órdenes. Cuando se introduzca el alias se sustituirá por el valor establecido.
`bg [job_spec]`	Reanuda en segundo plano la ejecución de un proceso parado con anterioridad.
`bind: bind [-lpvsPVS] [-m keymap] [-f filename] [-q name] [-u name] [-r keyseq] [keyseq:readline-function]`	Asocia una secuencia de teclas a una función Readline o a una macro.
`break [n]`	Pasa el control a la orden siguiente al bucle donde está incluido.
`builtin [shell-builtin [arg ...]]`	Ejecuta un shell interno.
`case WORD in [PATTERN [| PATTERN].`	Alternativa múltiple.
`cd [-PL] [dir]`	Cambia el directorio actual.
`command [-pVv] command [arg ...]`	Ejecuta la orden indicada ignorando las posibles funiciones definidas con igual nombre.
`continue [n]`	Pasa el control a la primera instrucción del bucle donde está incluido.
`declare [-afFrxi] [-p] name[=value] ...`	Declara variables y/o les da atributos.
`dirs [-clpv] [+N] [-N]`	Muestra la lista actual de directorios recordados.
`disown [-h] [-ar] [jobspec ...]`	Elimina el trabajo especificado de la lista de trabajos activos.
`echo [-neE] [arg ...]`	Muestra por pantalla los argumentos indicados.
`enable [-pnds] [-a] [-f filename]`	Habilita/deshabilita órdenes internas del shell. Útil cuando se quiere ejecutar un programa externo al shell con el mismo nombre que una orden interna sin tener que especificar el nombre completo del fichero donde se encuentra el programa.
`eval [arg ...]`	Evaluación de expresiones aritméticas.
`exec [-cl] [-a name] file [redir.]`	Ejecuta el programa especificado, sustituyendo el shell por ese programa.
`exit [n]`	Fin de ejecución del shell.

`export [-nf] [name ...] or export`	Marca los nombres indicados para ser exportados automáticamente al entorno de los subsiguientes programas que se ejecuten.
`false`	Devuelve fracaso.
`fc [-e ename] [-nlr] [first] [last] or fc -s [pat=rep] [cmd]`	Lista o edita y re-ejecuta órdenes de la historia de órdenes.
`fg [job_spec]`	Reanuda en primer plano la ejecución de un proceso parado con anterioridad.
`for NAME [in WORDS ... ;] do COMMANDS; done`	Realiza una iteración para cada uno de los valores tras el *in*, asignando en cada iteración el valor correspondiente a la variable NAME.
`function NAME { COMMANDS ; }`	Define una función que puede ser llamada por el nombre indicado.
`getopts optstring name [arg]`	Analiza los parámetros posicionales que se le pasan a un programa del shell.
`hash [-r] [-p pathname] [name ...]`	Recuerda el path completo de las órdenes indicadas por su nombre, de manera que para ejecuciones subsiguientes no tiene que buscarlas por los directorios de órdenes del sistema.
`help [pattern ...]`	Ofrece ayuda sobre órdenes internas del shell.
`history [-c] [n] or history -awrn [filename] or history -ps arg [arg...]`	Muestra y permite manipular la lista de órdenes que constan en la historia
`if COMMANDS; then COMMANDS; [ elif COMMANDS; then COMMANDS; ]... [ else COMM ANDS; ] fi`	Permite ejecución alternativa de órdenes en función del resultado de orden que sigue al if.
`jobs [-lnprs] [jobspec ...] or jobs -x command [args]`	Lista los trabajos que hay en la lista de trabajos. También permite añadir nuevos trabajos.
`kill [-s sigspec | -n signum | -sigspec] [pid | job]... or kill -l [sigspe c]`	Envía una señal a un proceso.
`let arg [arg ...]`	
`local name[=value] ...`	
`logout`	Termina la ejecución del shell..
`popd [+N | -N] [-n]`	Elimina entradas de la pila de directorios.
`printf format [arguments]`	Imprime con el formato especificado los argumentos dados.
`pushd [dir | +N | -N] [-n]`	Añade un directorio a la pila de directorios.
`pwd [-PL]`	Muestra el directorio actual.
`read [-r] [-p prompt] [-a array] [-e] [name ...]`	Lee una linea de la entrada estándar.
`readonly [-anf] [name ...] or readonly -p`	Marca las variables indicadas como de sólo lectura, sin que se pueda modificar su valor.
`return [n]`	Termina una función, devolviendo el control a la instrucción siguiente a su llamada.
`select NAME [in WORDS ... ;] do COMMANDS; done`	Muestra un menú con las opciones que se indican y recoge una opción por teclado.
`set [--abefhkmnptuvxBCHP] [-o option] [arg ...]`	Establece varias opciones en relación con variables, ejecución de órdenes, …
`shift [n]`	Desplaza n (ó 1, si no se especifica n) posiciones a la izda. los parámetros posicionales.
`shopt [-pqsu] [-o long-option] optname [optname...]`	
`source filename`	Lee y ejecuta órdenes del fichero, sin crear un nuevo shell, y vuelve.
`suspend [-f]`	Suspende la ejecución del shell hasta que se reciba la señal de continuación (SIGCONT).
`test [expr]`	Devuelve éxito si la expresión es verdadera y fracaso si no lo es.

`time [-p] PIPELINE`	Ejecuta una orden y muestra un resumen de distintos tiempos de ejecución de la orden.
`times`	Muestra el total de los tiempos de ejecución empleados por todos los procesos ejecutados desde el shell.
`trap [arg] [signal_spec ...] or trap -l`	Define una orden a ejecutar cuando se reciba una determinada señal.
`true`	Devuelve éxito.
`type [-apt] name [name ...]`	Muestra cómo se va a interpretar un nombre si se introduce como orden al shell.
`typeset [-afFrxi] [-p] name[=value] ...`	Similar a declare. Considerado obsoleto actualmente.
`ulimit [-SHacdflmnpstuv] [limit]`	Establece límites sobre el uso de distintos recursos por parte de los procesos ejecutados por el shell.
`umask [-p] [-S] [mode]`	Define la máscara de permisos de creación de ficheros.
`unalias [-a] [name ...]`	Elimina un alias anteriormente creado.
`unset [-f] [-v] [name ...]`	Elimina variables .
`until COMM1; do COMM2; done`	Repite las órdenes COMM2 hasta que COMM1 devuelva éxito.
`wait [n]`	Espera un número determinado de segundos.
`while COMM1; do COMM2; done`	Repite las órdenes COMM2 mientras COMM1 devuelve éxito.

Apéndice 2: Resumen de ordenes del shell

A continuación, se muestra una relación de las órdenes más habituales en Unix. La mayoría ya han sido tratadas con anterioridad, pero se incluyen aquí para tener una referencia rápida de su utilidad. Sólo se describe su funcionalidad básica; para mayor información sobre la misma y sobre los parámetros que se pueden incluir hay que acudir a las páginas del man correspondientes.

at	Permite definir tareas para que se ejecuten en un momento dado.
atq	Permite examinar la cola de tareas programadas para una ejecución posterior.
atrm	Permite examinar eliminar tareas de la cola de tareas programadas para una ejecución posterior.
banner	Genera letras grandes para imprimir carteles.
batch	Permite definir tareas para que se ejecuten en un momento dado, siempre y cuando la carga del sistema no sea excesiva.
cal	Muestra el calendario del mes actual o del que se especifique como parámetro.
calendar	Muestra los eventos definidos en el sistema para los próximos días
cat	Concatena ficheros y los muestra en la salida estándar
chgrp	Cambia el grupo al que pertenece un fichero
chmod	Cambia los permisos de acceso a un fichero
chown	Cambia el propietario de un fichero
clear	Limpia la pantalla
cp	Copia ficheros
crontab	Permite programar ejecuciones periódicas o puntuales de programas
date	Muestra la fecha y hora del sistema. Con privilegios adecuados, permite establecer la fecha y hora del sistema
echo	Muestra una línea de texto en la salida estándar.
exit	Termina la ejecución del programa que lo ejecuta.
grep	Muestra las líneas de ficheros que casan con un patrón determinado.
head	Muestra las primeras líneas de un fichero (o ficheros).
kill	Envía una señal a un proceso. Su utilidad más habitual es enviar la señal 9 ó la 15 para terminar con un proceso
last	Muestra la relación de los últimos usuarios que han accedido al sistema.
less	Permite visualizar por pantalla un fichero, pudiendo avanzar y retroceder para verlo mejor.
ln	Crea nombres alternativos (enlaces) para ficheros y directorios.
login	Crea una sesión en el sistema.
logname	Muestra el nombre de acceso al sistema del usuario.

ls	Lista el contenido del directorio.
mail	Envía y recibe correo electrónico.
man	Sirve para acceder al manual del sistema.
mesg	Escribe directamente en una determinada terminal.
mkdir	Crea directorios
more	Visualizador de ficheros, que para la salida cuando se llena una pantalla.
mv	Renombra ficheros
newgrp	Cambia el grupo activo del usuario.
nice	Ejecuta un programa con baja prioridad
nl	Muestra un fichero por pantalla, numerando sus líneas.
passwd	Cambia la password del usuario.
ps	Lista los procesos actualmente en ejecución.
pwd	Muestra el directorio actual del usuario.
rm	Borra ficheros y directorios
rmdir	Elimina directorios vacíos.
seq	Genera un secuencia de números.
sleep	Retarda la ejecución del proceso un número determinado de segundos.
sort	Ordena las líneas de ficheros de texto.
su	Cambia la identidad del usuario, normalmente para convertirse en superusuario.
tail	Muestra las últimas líneas de un fichero
top	Muestra, de manera interactiva, los procesos en ejecución.
tty	Muestra el nombre de la terminal que estamos utilizando.
umask	Establece la máscara de permisos en la creación de ficheros.
uname	Muestra información del sistema (nombre, versión del SS.OO., ...)
uptime	Muestra el tiempo que lleva el sistema en funcionamiento, así como su carga actual
wall	Envía un mensaje a las pantallas de todos los usuarios conectados.
wc	Cuenta el número de caracteres, palabras y líneas de un fichero.
who	Muestra los usuarios que están conectados en el sistema.
write	Envía un mensaje a la pantalla de otro usuario