Aprendiendo Linux: Introducción al SO Linux

Introducción (entrada al sistema, salida de sistema, introducción de órdenes, algunas órdenes útiles, el Shell).

Tabla de Contenidos

Autor: Lina García Cabrera & Francisco Martínez del Río

1 Entrada al sistema

El sistema LINUX es un entorno de computación multiproceso, multiusuario e interactivo. **Multiproceso** significa que en un instante dado se pueden ejecutar varios procesos o programas a la vez. **Multiusuario** significa que muchos usuario pueden estar usando el sistema a la vez.

En un sistema operativo multiusuario es vital que el sistema pueda **conocer y diferenciar** a los distintos usuarios activos en un instante dado. Eso se consigue mediante la conexión (*login*) y desconexión (*logout* o *logoff*) del sistema.

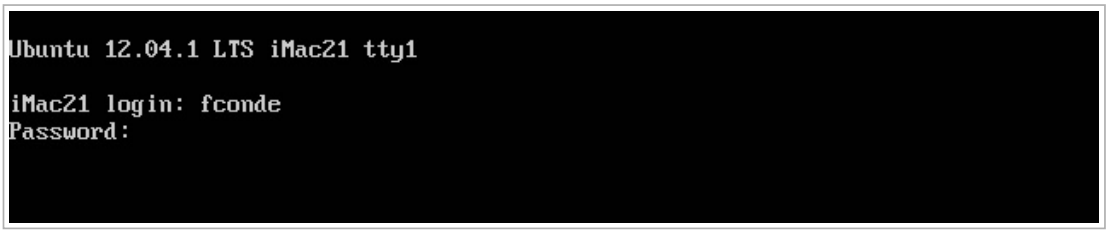
Para que un usuario pueda identificarse en el sistema y ganar acceso al mismo, debe tener una **cuenta de usuario** activa definida en el sistema. Una cuenta de usuario es información sobre el usuario que el sistema almacena y una carpeta en el disco propiedad del usuario.

Las cuentas de usuario las administra (crea, modifica y elimina) el administrador del sistema, que es un usuario con permisos especiales que se encarga de gestionar el sistema operativo.

Cuando un usuario quiere acceder a un sistema operativo multiusuario, el sistema le pide sus **credenciales de acceso.** Las credenciales de acceso son un **nombre de usuario** (*login*) y una **clave de acceso** (*password*). Si el sistema operativo ha arrancado en modo consola (en Ubuntu Linux es el modo 3), o bien nos hemos conectado a él desde otro ordenador mediante conexión remota, el sistema nos pedirá en la pantalla el nombre de usuario escribiendo el mensaje:

> `[nombre ordenador] login:`

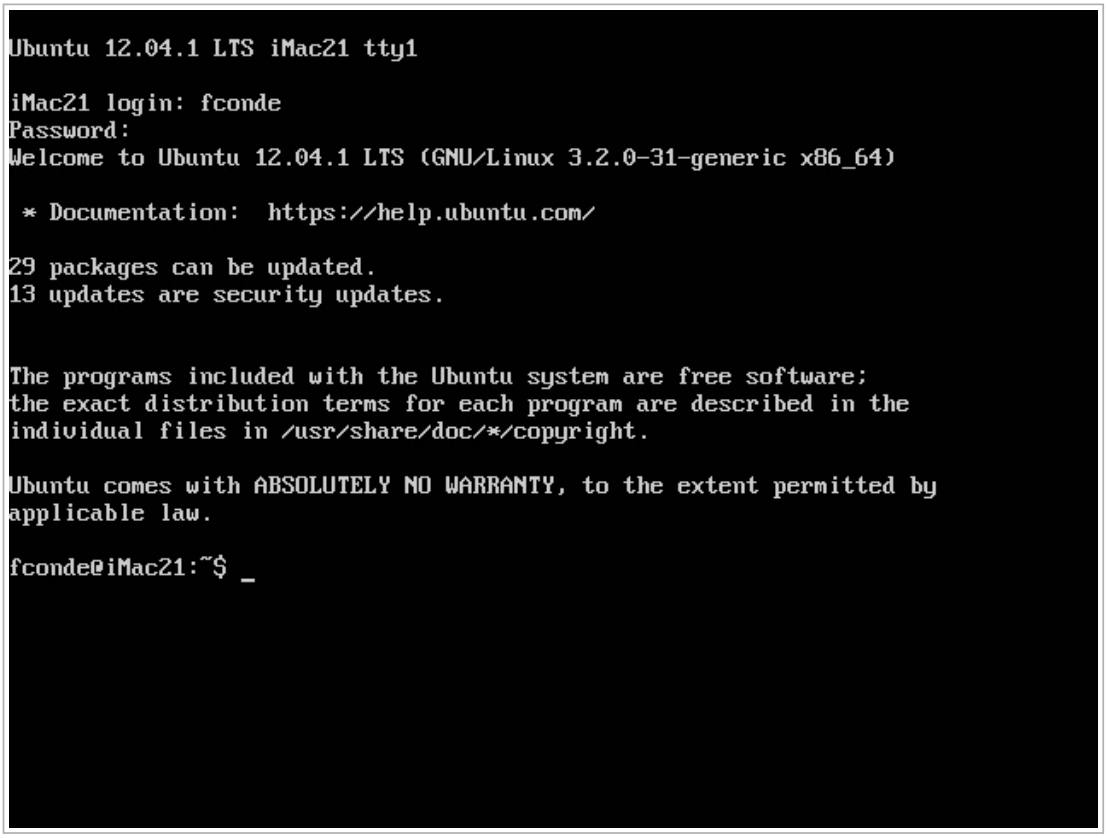
Una vez introducido el nombre de usuario hay que pulsar *Enter*, y el sistema pregunta la clave de acceso. Hay que teclearla y pulsar *Enter*. Mientras se teclea la clave, ésta no se muestra en pantalla para evitar que alguien pudiera verla y acceder al sistema suplantando nuestra identidad.



Es muy importante darse cuenta de que **UNIX distingue entre mayúsculas y minúsculas** (es *case sensitive*), por lo que hay que tener cuidado al teclear la clave.

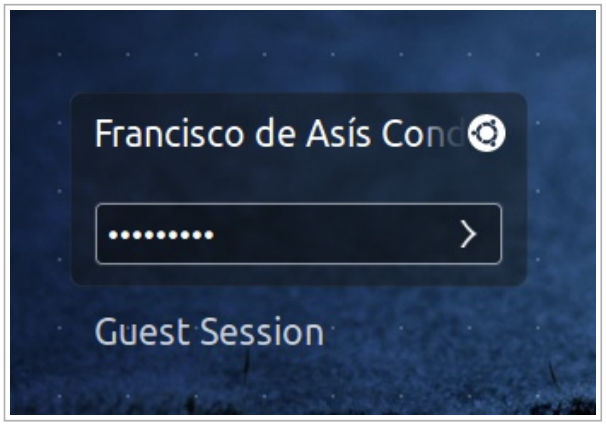
> Si la clave es `Woody.Al`, no se admitirá como clave `woody.al` o `WOODY.AL`.

Si tecleamos correctamente las credenciales de acceso, habremos iniciado una **sesión** con la máquina y se nos mostrará un mensaje de bienvenida y un indicador (*prompt*) que indica que el **intérprete de órdenes** (*shell*) está listo para que introduzcamos alguna orden. El indicador depende del intérprete de órdenes concreto que se esté usando, siendo indicadores típicos el signo de dólar (**$**, para la shell llamada **`bash`** *Bourne Again Shell*) o el signo de porcentaje (**%**, para la shell **`tcsh`** *Tenex style C Shell*).



Cuando un administrador crea una cuenta de usuario asigna una clave de acceso por defecto. El usuario puede cambiarla cuando lo desee ejecutando la orden `passwd` en el intérprete de órdenes. Esta orden preguntará la clave de acceso actual y luego solicitará dos veces la clave nueva, la segunda de ellas para confirmar que está bien escrita.

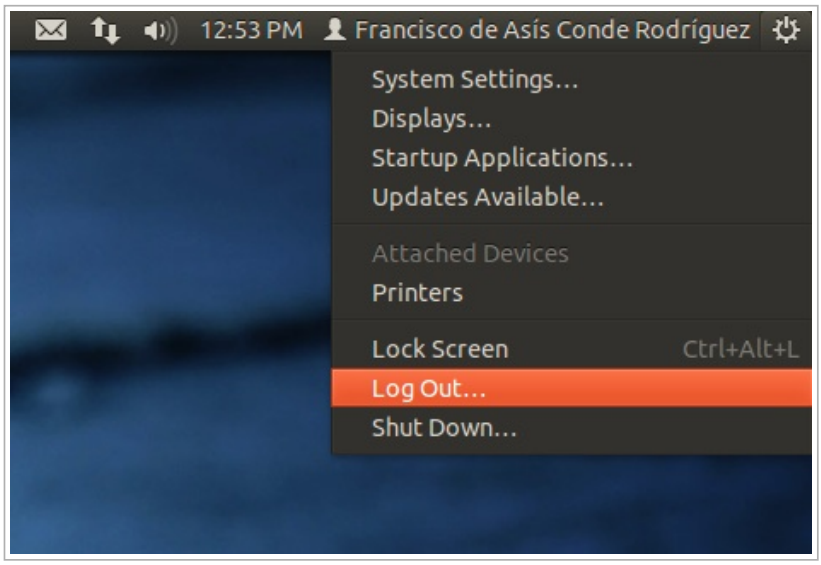
Si el sistema operativo ha arrancado en modo interfaz gráfica de usuario (en Ubuntu Linux es el modo 2), el proceso de solicitar las credenciales se realiza en una ventana gráfica.



2 Salida del sistema

Si quiere terminar la sesión y salir del sistema, pruebe a teclear ***exit***, ***logout***, o pulse ***ctrl-d***. Alguno de estos métodos funcionará en su sistema. La salida vía ***ctrl-d*** muchas veces se desactiva para evitar desconexiones accidentales.

Si se ha arrancado Ubuntu Linux en modo de interfaz gráfica interactiva, la desconexión del sistema se realiza mediante la orden `Log Out…` del menú principal.



3 Introducción de órdenes

Una vez que el sistema muestra el indicador, se pueden introducir **órdenes** o **mandatos** (*commands*).

La sintaxis general o estructura de una orden es la siguiente:

$ orden [[-]opcion(es)] [argumento(s) de opcion(es)] [argumento(s) de orden(es)] donde:

$ es el indicador de petición de orden de la shell,

cualquier cosa encerrada entre [] es opcional,

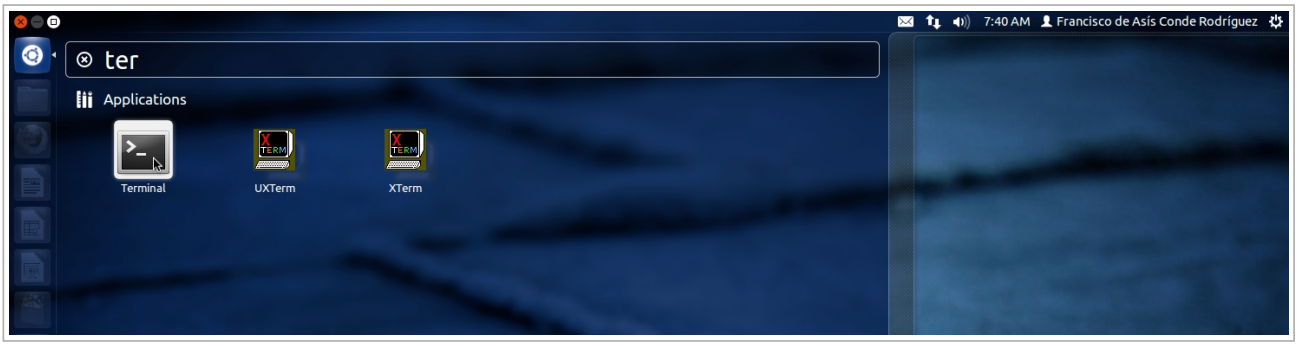
***orden** es el nombre de una orden LINUX válida para ese shell, escrita con letras minúsculas,*

[[-]opcion(es)] *es uno o más modificadores que cambian el comportamiento de la orden,*

[argumento(s) de opcion(es)] *es uno o más modificadores que cambian el comportamiento de las opciones, y*

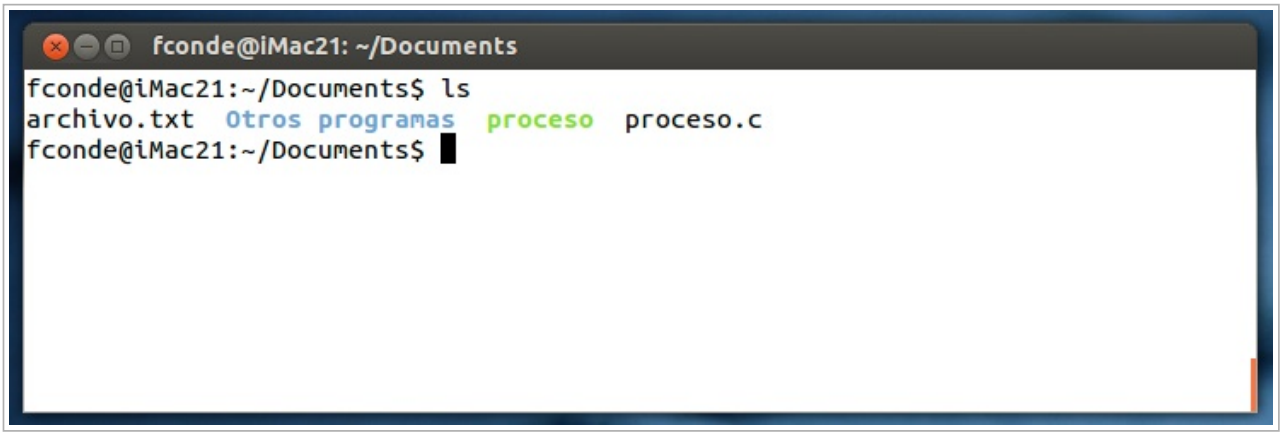
[argumento(s) de orden(es)] *es uno o más objetos, separados por espacios, que serán afectados por la orden.*

Para iniciar un terminal en el caso de que Ubuntu Linux haya arrancado en modo interfaz gráfica interactiva, se usa el ***dash*** (primer icono de la paleta de la izquierda). En el espacio para buscar se escriben las primeras letras de la palabra ***Terminal*** (en este caso no hace falta distinguir entre mayúsculas y minúsculas) y el dash encontrará la aplicación Terminal, que es la forma de invocar al intérprete de órdenes en Ubuntu Linux.



Para introducir una orden se escribe el nombre de la orden seguida de cero o más argumentos separados por espacios en blanco y/o tabuladores, y seguido de la pulsación de la tecla *Enter*.

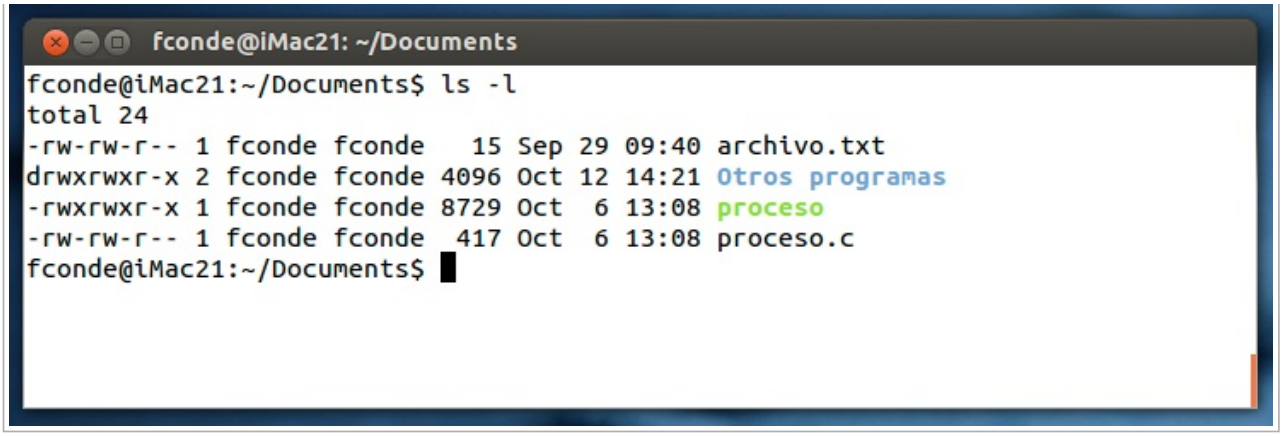
Por ejemplo, la orden **ls** (abreviatura de *list*) sirve para listar los ficheros y directorios que cuelgan de los directorios. Si la utilizamos sin argumentos lista o enumera los ficheros y directorios que cuelgan del **directorio de trabajo** (el directorio en el que nos encontramos situados) ordenados lexicográficamente.



Pasemos a comentar algo sobre los **argumentos** o **parámetros de las órdenes**. Existen dos tipos de parámetros: **los fijos y los variables**.

- Los **parámetros fijos** siempre toman el mismo valor, indican el modo de funcionamiento que se desea de la orden. En la mayoría de los casos van después de la orden, precedidos por el símbolo del guión (-), por ejemplo:

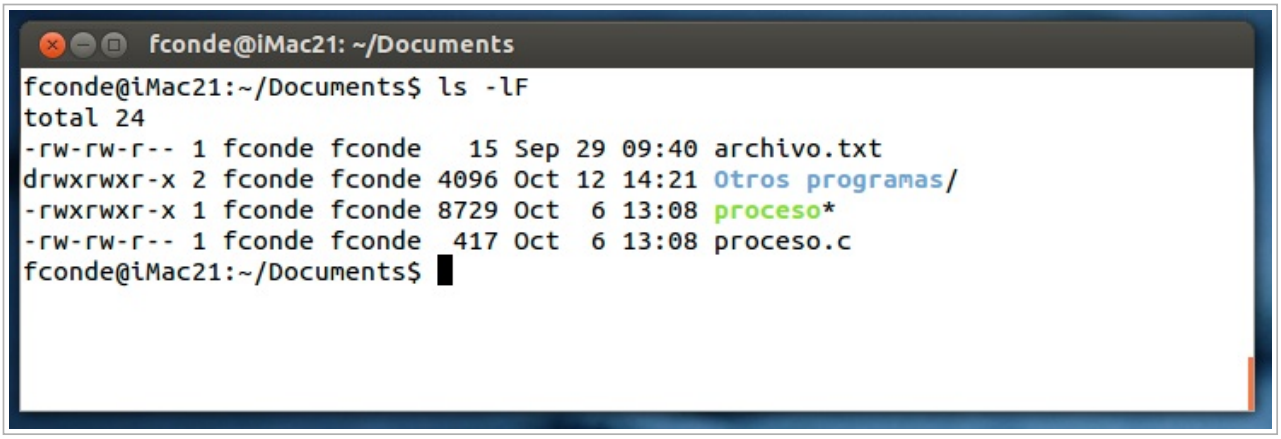
La opción **l** produce un listado más completo (*long*), con un línea por ítem, en cada línea se especifican datos sobre cada ítem. Si se quiere utilizar varios parámetros fijos en una orden se puede poner un guión y poner todas las opciones seguidas, o preceder cada opción por un guión. Por ejemplo, las órdenes **ls -lF** y **ls -l -F** son equivalentes y producen el listado:



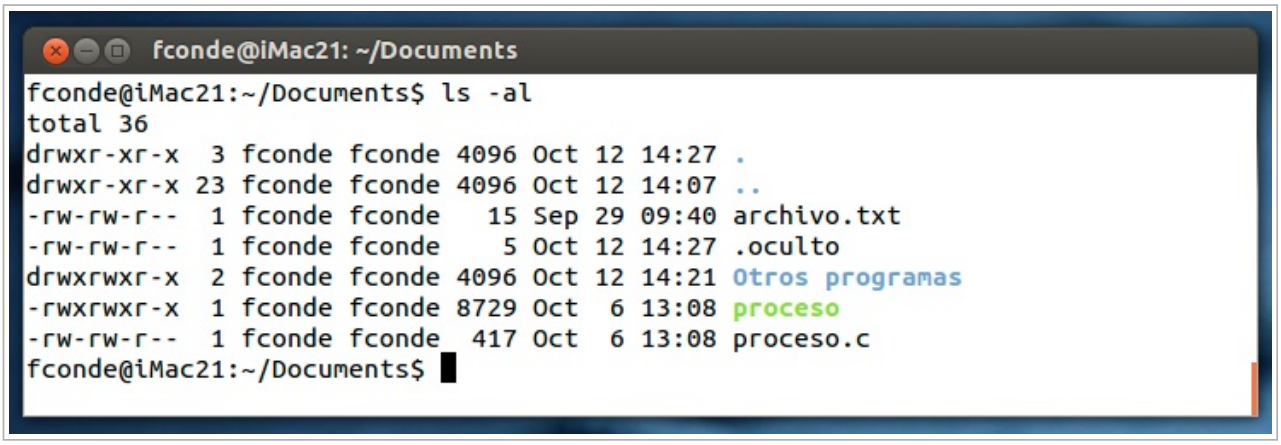
La opción **F** coloca una diagonal '/' después de cada fichero que es un directorio, y un asterisco '*' después de cada fichero ejecutable.

- Por último, están los **parámetros variables**, como su nombre indica su valor no es fijo.

Así, la orden **ls** admite como parámetro un nombre de directorio, **ls** listará todos los ficheros que cuelguen de dicho directorio. Por ejemplo, **ls -l /bin** producirá un listado largo de todos los ficheros que cuelgan del directorio */bin*.



La opción **a** (*all*) hace que aparezcan en el listado los **archivos ocultos** (su nombre comienza por **.**)



Uno de los defectos de LINUX es la **falta de consistencia en los argumentos de las órdenes**.

- Así, en la mayoría de las órdenes los **argumentos fijos pueden ponerse antes o después de los variables**, pero hay órdenes en las que esto no ocurre.
- Otro inconveniente aparece cuando se utilizan varios argumentos fijos en una orden; **algunas órdenes permiten especificarlos todos tras un guión o utilizar un guión por cada argumento fijo; otras órdenes, sin embargo, sólo permiten una de las opciones**.
- Por último, **algunas letras utilizadas para especificar argumentos fijos representan funciones distintas en órdenes distintas**; por ejemplo, -r significa la utilización recursiva de ciertas órdenes, mientras que otras, que no incluyen la opción –r, su utilización recursiva se especifica mediante -s.

4 Algunas órdenes útiles

En este apartado se van a comentar algunas de las órdenes que se utilizan con mucha frecuencia, de este modo el lector curioso podrá inspeccionar los ficheros del sistema antes de que estudiemos con más rigor estas órdenes.

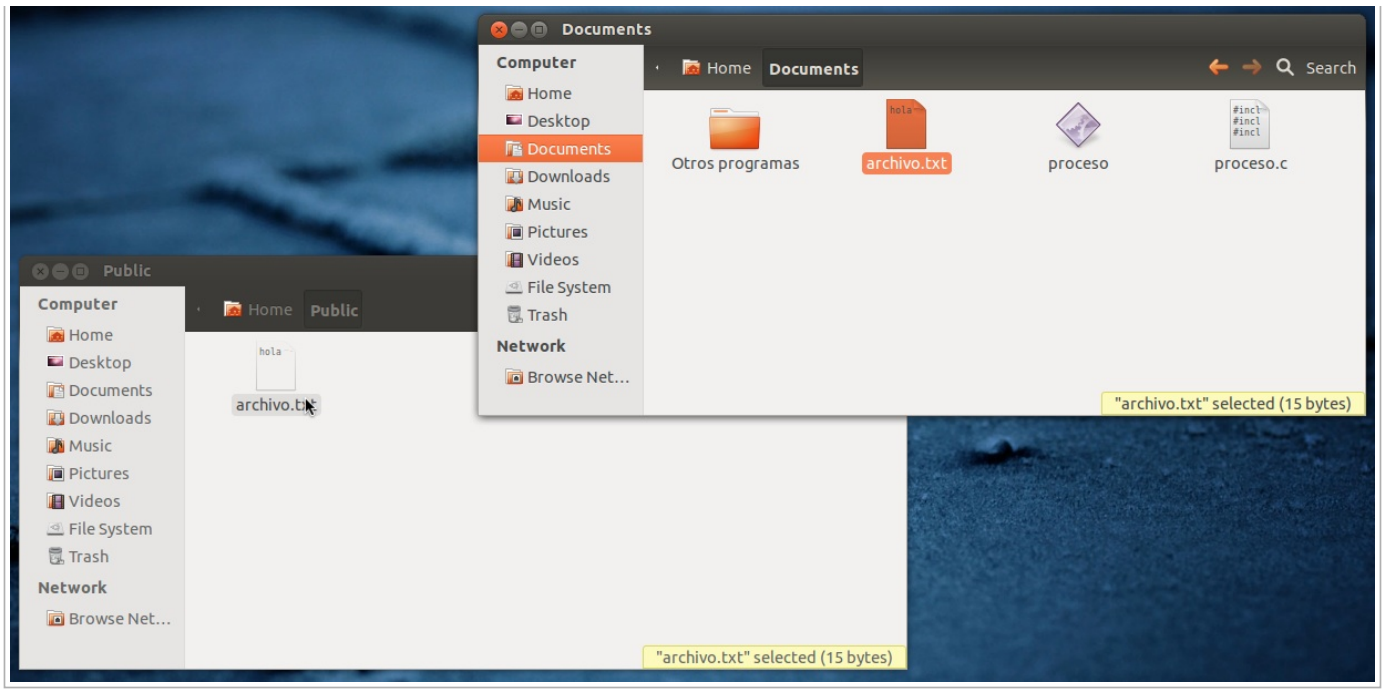
- **pwd**: muestra el directorio de trabajo (Print Working Directory).
- **cd [*dir*]**: sirve para cambiar el directorio (Change Directory) de trabajo. Si se utiliza sin argumentos cambia el directorio de trabajo al directorio base (home directory). Hay que tener en cuenta que en UNIX se usa la diagonal ('/'), en lugar de la diagonal invertida ('\') utilizada en Windows, en las trayectorias (*paths*) de los ficheros. La trayectoria \ab\c de Windows se escribe /ab/c en UNIX.
- **more fich**: el programa more es un paginador, es decir, un programa que muestra las páginas de un fichero de texto. Los paginadores no están normalizados, por tanto, es posible que en su sistema no tenga el programa **more** (si es así, pruebe a ver si existe el paginador **pg**). Si el texto del fichero no cabe en la pantalla, pulsando *Enter* se puede ver la siguiente línea del fichero, y pulsando el espaciador una pantalla más del fichero. Una vez que ha ejecutado more puede obtener una ayuda si pulsa la letra **h**.
- **cp**: sirve para copiar (*copy*) ficheros.
- **mv**: sirve para desplazar (*move*) un fichero en el sistema de ficheros.
- **rm**: sirve para borrar -*remove*- ficheros (para ser exactos borra enlaces como veremos más adelante).
- **man *orden***: UNIX tiene un manual para consultarlo en línea desde el terminal del ordenador. Teclee **man cp** y obtendrá información sobre la orden **cp**. Escriba **man man** y obtendrá ayuda sobre el manual. El manual también incluye información sobre llamadas al sistema, rutinas de biblioteca de C, juegos, y más cosas.
- **whatis *orden/es***: con esta orden se puede obtener una descripción abreviada de lo que hace cualquier orden. Se pueden pasar más de un argumento en la misma línea de órdenes separadas por espacios para consultar varias órdenes.
- **whoami**: muestra en pantalla nuestra identidad de usuario.
- **hostname**: muestra el nombre del computador al que nos hemos conectado.
- **uname**: muestra información relacionada con el sistema operativo que se ejecuta en el computador.
- **wc**: lee la entrada estándard y cuenta caracteres (wc -c), palabras (wc -w) o líneas (wc -l).

5 El Shell

Cuando el sistema ha arrancado en modo consola y se efectúa la conexión, el sistema LINUX pone en marcha **un programa que actúa como interfaz entre nosotros y el sistemas operativo**. En consecuencia, cuando el sistema despliega el indicador '$' y se teclean órdenes que son ejecutadas por el sistema, no es el sistema operativo el que se está dirigiendo al usuario, sino un intermediario llamado **intérprete de órdenes** o *shell*.

Intérprete de órdenes, **Intérprete de mandatos**, **Intérprete de línea de mandatos**, **Intérprete de comandos**, **Terminal**, **Consola**, **Shell** ó su acronimo en idioma inglés **CLI** por **Command line interface**, es un programa informático que actúa como Interfaz de usuario para comunicar al usuario con el sistema operativo mediante una ventana que espera ordenes escritas por el usuario en el teclado (por ej. sort -n listatelefonos > listatelefonos.ordenada), los interpreta y los entrega al sistema operativo para su ejecución. La respuesta del sistema operativo es mostrada al usuario en la misma ventana. A continuación, El programa *shell* queda esperando más instrucciones.

Cuando el sistema ha arrancado en modo interfaz gráfica interactiva, la propia interfaz permite comunicación entre usuario y sistema operativo. Así, si un usuario arrastra (con la tecla *Ctrl* pulsada) el icono de un documento de texto llamado `Archivo.txt` desde la carpeta `~/Documents` a la carpeta `~/Public`, en realidad está llamado a la orden: **`cp ~/Documents/Archivo.txt ~/Public`** La interfaz gráfica interpreta las acciones del usuario con el ratón sobre los iconos del sistema y la traduce a una orden para el sistema.



5.1 El shell como ejecutor de órdenes

Cuando el ***shell*** despliega el indicador, el usuario teclea una orden (un programa), seguida de sus argumentos. Una de las misiones del *shell* es realizar las llamadas al sistema oportunas para ejecutar la orden tecleada, una de ellas lo convierte en **proceso**. Para ilustrar esto, a continuación se muestra el código de un programa que representa un *shell* muy simplificado:

```
 1 #include <stdlib.h>
 2 #include <string.h>
 3 #include <stdio.h>
 4 #include <unistd.h>
 5 #include <sys/types.h>
 6 #include <sys/wait.h>
 7 #define MAXLINE 100
 8
 9 int main(void)
10 {
11         char buf[MAXLINE];
12         id_t pid;
13         int status;
14
15         printf("%% "); /* escribe el indicador % (printf requiere %% para escribir %) */
16
17         while (fgets(buf, MAXLINE, stdin) != NULL) {
18
19                 buf[strlen(buf) - 1] = '\0'; /* cambia newline por nulo */
20
21                 pid = fork();
22
23                 if (pid < 0) { /* no se pudo crear el hijo */
24
25                         puts("Error en fork");
26                         exit(127);
27
28                 } else if (pid == 0) { /* hijo */
29
30                         execlp(buf, buf, (char *) 0);
31                         printf("No se pudo ejecutar %s\n", buf);
32                         exit(126);
33
34                 } else { /* padre */
35
36                         if ((pid = waitpid(pid, &status, 0)) < 0) {
37                                 puts("Error en waitpid");
38                                 exit(125);
39                         }
```

Para poder ejecutar ese shell simplificado, en primer lugar hay que escribirlo o copiarlo a un fichero con nombre shell.c. A continuación hay que compilarlo y enlazarlo, esto se realiza mediante la orden **`gcc shell.c -o shell`**. Por último hay que ejecutarlo mediante la orden **`./shell`**.

En la siguiente imagen puede verse cómo compilar y ejecutar el programa en un Terminal de Ubuntu Linux (en la última línea se ha pulsado **Ctrl-d**):

Tal vez al intentar ejecutar el programa te has encontrado con el problema "**orden no encontrada**" o "**command not found**"

¿Por qué al ejecutar una orden en el shell me dice "orden no encontrada" o "command not found"?

```
40                          printf("%% ");
```

```
fconde@iMac21: ~/Documents/SO
fconde@iMac21:~/Documents/SO$ gcc shell.c -o shell
fconde@iMac21:~/Documents/SO$ ./shell
% ls
shell  shell.c
% whoami
fconde
% pwd
/home/fconde/Documents/SO
% ls -lF
No se pudo ejecutar ls -lF
% fconde@iMac21:~/Documents/SO$
```

A continuación se comentan ciertos aspectos del programa:

- [Línea 17] Se utiliza la función de E/S estándar `fgets` para leer una línea de una vez de la entrada estándar. Cuando se escribe el carácter fin de fichero (o carácter **EOF**, del inglés **E**nd **O**f **F**ile), el cual normalmente es `ctrl-d`, como el primer carácter de una línea, `fgets` devuelve un puntero nulo, el ciclo se detiene, y el proceso finaliza.
- [Línea 19] Debido a que las líneas devueltas por `fgets` terminan con un carácter salto de línea (*newline*), seguido por un byte nulo, se utiliza la función del C estándar `strlen` para calcular la longitud de la cadena de caracteres, y se sustituye el carácter *newline* por un byte nulo. Esto se hace así porque la función `execlp` requiere que sus argumentos finalicen en un carácter nulo, y no que terminen con el carácter *newline*.
- [Línea 21] Se llama a **`fork`** para **crear un proceso nuevo**. El nuevo proceso es una copia del proceso llamador, y se dice que el proceso que llama es el padre y el proceso recién creado es el hijo. **`fork`** devuelve el identificador de proceso (un entero no negativo) del nuevo proceso hijo al padre, y devuelve 0 al hijo. Puesto que **`fork`** crea un nuevo proceso, se dice que es llamado una vez (por el padre) y que devuelve un valor dos veces (tanto en el padre como en el hijo).

  Cuando se hace la llamada a **`fork`**, se copia el proceso en el ESTADO en el que se encuentran. Por tanto, tanto padre como hijo retoman la ejecución (cuando el SO le asigne la CPU, eso ocurrirá cuando decida el planificador) en la asignación a la variable **`pid`**. La variable **`pid`** ocupará dos casillas de memoria distintas pertenecientes al espacio de memoria en un caso del proceso padre y en el otro del proceso hijo.
- [Línea 30] En el hijo se llama a **`execlp`** para ejecutar la orden que fue leída de la entrada estándar. Esto reemplaza el código del proceso hijo por el fichero del nuevo programa. La combinación de un **`fork`**, seguido de un *exec* (*execelp* es un caso particular de la llamada al sistema *exec*) es lo que algunos sistemas operativos llaman intercambio de un nuevo proceso. En UNIX las dos partes se separan en funciones individuales.

  Es muy importante darse cuenta de que **`execlp` reemplaza** al proceso que la llama (el proceso hijo), por tanto sólo si falla se ejecutan las líneas 31 y 32. En caso contrario, esas líneas nunca se ejecutan.
- [Línea 36] Ya que el hijo llama a **`execlp`** para ejecutar el fichero del nuevo programa, el padre necesita esperar a que termine el hijo. Esto se hace llamando a **`waitpid`**, especificando a qué proceso deseamos esperar (el argumento ***pid***, que es el identificador de proceso del hijo). La función **`waitpid`** también devuelve el estado de terminación del hijo (en la variable ***status***), pero en este *shell* tan simple no se hace nada con su valor. Se podría examinar para determinar cómo terminó el hijo.
- [Líneas 23 a 41] La llamada a la función **`fork()`** hace una copia del proceso que la llama. Por tanto se tienen dos procesos cuyas instrucciones de programa son idénticas. Si se quiere que realicen tareas distintas, la única solución es introducir instrucciones de programa que averiguen de qué proceso se trata, y en función de si es el padre o el hijo, que realicen una determinada tarea u otra.

  Esta es la misión del **`if - then - else`**.

  La primera parte del **`if`** pregunta si pid es menor que cero. Si eso pasa es porque la llamada a **`fork()`** no ha podido crear el subproceso.

  La segunda parte del **`if`** pregunta si pid es cero. Si lo es, es porque esa instrucción la está ejecutando el subproceso o proceso hijo. Recuerda que **`fork()`** devuelve cero al subproceso.

La tercera parte del if no necesita preguntar nada, ya que pid sólo puede ser mayor que cero. Esto indica que esa instrucción la está ejecutando el proceso padre. **`fork()`** devuelve al proceso padre el id del subproceso que crea.

- La limitación fundamental de este programa es la de que el usuario no puede pasar argumentos a la orden que desea ejecutar. No se puede, por ejemplo, especificar el nombre de un directorio a listar. Solamente se puede ejecutar **ls** sin argumentos (por lo tanto, **ls** actuará sobre el directorio de trabajo). Para permitir argumentos se requeriría analizar sintácticamente la línea de entrada, separándolos, y pasándole cada argumento de la orden como un argumento separado a la función *execlp*.

Comprender cómo funciona un shell por dentro no es fácil sobre todo si no se entiende qué es un proceso. Puede que sigas preguntándote ¿para qué sirven las llamadas `fork` y `wait`?

5.2 El shell como intérprete de metacaracteres

El *shell* que hemos visto en el apartado anterior era muy simple, una primera mejora ya se apuntó más arriba, consiste en analizar la línea de entrada para pasarle los argumentos a las órdenes. Sin embargo, antes de pasar los argumentos a las órdenes el propio *shell* realiza una interpretación de ciertos caracteres de la línea de entrada. Tras realizar dicha interpretación procede a pasarle los argumentos (posiblemente distintos a los originales debido a la interpretación) a la orden. Los caracteres que el *shell* interpreta se llaman **metacaracteres**. A continuación vamos a estudiar dos tipos de metacaracteres que interpretan todos los *shells* importantes, en concreto son:

- Los **metacaracteres mágicos o comodines**. Sirven para especificar un conjunto de ficheros indicando solamente el patrón de sus nombres. El shell encontrará los ficheros que verifiquen el patrón.
- Los **metacaracteres de redireccionamiento de la E/S**. Se puede lograr que la salida de cualquier programa se dirija hacia un fichero, en lugar de dirigirse hacia el terminal; se puede hacer que la entrada provenga también de un fichero y no del terminal. La entrada y la salida incluso pueden conectarse a otros programas.

5.2.1 Abreviaturas de nombres de ficheros

Comencemos con los patrones de nombres de ficheros. Suponga que se está tecleando un documento grande, por ejemplo un libro. Éste se divide en partes de menor tamaño, como capítulos y, tal vez, secciones. También físicamente habrá de estar dividido, debido a la incomodidad que representa la edición de ficheros extensos. Así, el documento debe ser tecleado en varios ficheros. Se podrían tener ficheros separados para cada capítulo, llamados cap1, cap2, etc. Asimismo, si cada capítulo fuera dividido en secciones, podrían crearse los siguientes ficheros:

> cap1.1
>
> cap1.2
>
> cap1.3
>
> . . .
>
> cap2.1
>
> cap2.2

Es conveniente que ejecute el siguiente programa (se lo puede bajar a su directorio de trabajo desde el recurso “preguntas frecuentes” de la carpeta “Materiales de prácticas”) o desde aquí:

Fichero/programa en bash
creacap (154 B)

Descarga el fichero **creacap** y cambia sus permisos con **chmod +x creacap** para hacerlo ejecutable y ejecútalo.

> **$ creacap**

y se crearán en su directorio 30 ficheros de nombre *capi*, cap*i*.1 y cap*i*.2 para *i* desde 1 hasta 10. El contenido de los ficheros capx y capx.y es la cadena de caracteres capítulox y capx.y respectivamente.

Con una nomenclatura sistemática se puede saber rápidamente dónde encaja un fichero dentro del total. ¿Y si el usuario quisiera ver el contenido de todo el libro? Podría teclear

> **$ cat cap1.1 cap1.2 cap1.3...**

ya que la orden cat tiene como efecto mostrar en la pantalla el contenido de los ficheros que se le pasen como argumentos. Sin embargo, el usuario pronto se aburriría de teclear tantos nombres de fichero y empezaría a cometer errores. Aquí es donde entran en acción las abreviaturas de nombres de fichero. Si se teclea

> **$ cat cap***

el **shell** toma el * como "cualquier cadena de caracteres", de modo que *cap** es un patrón que corresponde a todos los nombres

de fichero en el directorio de trabajo que empiecen con *cap*. El *shell* crea la lista, en orden alfabético (para ser exactos en el orden en que vienen escritos los caracteres en el código ASCII) y la pasa a **cat**. La orden **cat** nunca ve el *. La búsqueda que hace el *shell* en el directorio de trabajo genera una lista de cadenas que se pasan a **cat**.

El punto crucial es que la abreviatura del nombre de fichero no es propiedad de la orden **cat**, sino un servicio del *shell*. Por ello, puede utilizarse para generar una secuencia de nombres de ficheros destinados a cualquier orden. Existe un programa llamado **echo** que es especialmente útil para ensayar el significado de los comodines. Como puede suponerse, **echo** no hace otra cosa que retransmitir ("hacer eco a") sus argumentos al terminal:

```
$ echo hello world

hello world
```

Pero los argumentos pueden ser generados por medio de patrones:

```
$ echo cap1.*
```

lista los nombres de todos los ficheros del capítulo 1,

```
$ echo *
```

lista por orden alfabético todos los nombres de ficheros que se encuentren en el directorio de trabajo,

```
$ cat*
```

muestra en la pantalla el contenido de todos los ficheros del usuario (en orden lexicográfico), y

```
$ rm *
```

borra todos los ficheros del directorio de trabajo (se debe estar muy seguro de que esto es lo que se desea hacer). El * no está limitado a la última posición en un nombre de fichero. Se pueden colocar uno o varios de ellos en cualquier posición del nombre de fichero. Así,

```
$ rm *.save
```

elimina todos los ficheros que terminan en *.save*.

Observe que los nombres de fichero se ordenan lexicográficamente, lo cual no es lo mismo que si estuvieran ordenados numéricamente. Si el libro tiene 10 capítulos, el orden tal vez no sea el deseado por el usuario, ya que cap10 queda antes que cap2:

```
$ echo *

cap1 cap10 cap10.1 cap10.2 cap1.1 cap1.2 cap2 cap2.1 cap2.2 ...
```

El * no es la única característica de reconocimiento de patrones del *shell*, aunque sea la utilizada más frecuentemente. El patrón [...] representa cualquiera de los caracteres dentro de los corchetes. Podemos abreviar un rango de letras o dígitos consecutivos:

```
$ cat cap[12346789]*      Mostrar los capítulos 1, 2, 3, 4, 6, 7, 8, y 9, pero no el 5

$ cat cap[1-46-9]*        Lo mismo que antes

$ rm temp[a-z]            Elimina los tempa, ..., tempz que existan
```

El patrón ? representa a cualquier carácter:

```
$ ls ?                    Mostrar los ficheros con nombre de un sólo carácter

$ ls -l cap?.1            Mostrar cap1.1 cap2.1 cap3.1 etc, pero no cap10.1

$ rm temp?                Eliminar los ficheros temp1, ..., tempa, etc.
```

Nótese que estos patrones funcionan sólo con nombres de ficheros existentes (pues el *shell* iguala el patrón con ficheros que existen). En particular, no es posible crear nuevos ficheros usando patrones. Por ejemplo, si se desea expandir *cap* a capítulo en cada nombre de fichero no puede hacerse de la siguiente manera:

```
$ mv cap.* capítulo.*     ¡No funciona!
```

ya que *capítulo.** no concuerda con ningún fichero existente. Esto sí se puede hacer en MS DOS con la orden **move** ya que el intérprete de órdenes de MS DOS (el *command.com*) no interpreta el *, sino que le pasa a la orden **move** como argumentos *cap.** y *capítulo.**, siendo **move** la encargada de interpretar el asterisco.

Los caracteres de patrones tales como * pueden usarse tanto en trayectorias como en nombres de ficheros simples; la búsqueda se hace para cada componente de la trayectoria que contenga un metacarácter *. Así, */usr/mary/** lleva a cabo la búsqueda dentro de */usr/mary*, y */usr/*/calendar* genera una lista de nombres de todos los ficheros calendar que cuelgan de los subdirectorios de */usr*.

Si alguna vez desea eliminar el significado especial de *, ?, etc., deberá encerrarse el argumento completo entre apóstrofos, como en

```
$ ls '?'
```

esto también puede hacerse anteponiendo al metacarácter una diagonal invertida:

> **$ ls \?**

ambas órdenes listarían un fichero cuyo nombre fuera ?, si existe uno con tal nombre.

5.2.2 Redireccionamiento de la E/S

> En el sistema operativo UNIX toda la entrada/salida se realiza leyendo o escribiendo en ficheros, ya que todos los periféricos, incluso los terminales, son ficheros del sistema de ficheros. Esto significa que una interfaz sencilla y homogénea gestiona toda la comunicación entre un programa y los periféricos.

Cuando empieza la ejecución de un programa en UNIX éste se inicia con **tres ficheros abiertos**: la **entrada estándar**, la **salida estándar** y la **salida de error estándar** (algunas veces llamada error estándar). Estos tres ficheros están asociados por defecto al terminal, por lo que la lectura de la entrada estándar equivale a leer del teclado y la escritura en la salida y el error estándar equivale a escribir en la pantalla. El lenguaje C proporciona una biblioteca estándar de E/S, esta biblioteca suministra una serie de funciones que leen y escriben en los ficheros antes mencionados (también proporciona funciones para hacer otras cosas). Para utilizar la biblioteca estándar de E/S hay que añadir la línea

```
1 #include <stdio.h> /* stdio viene de STanDard Input/Output */
```

al principio del fichero fuente que utilice las funciones de la biblioteca. El uso de los signos de relación (<, >) instruye al compilador para buscar el fichero en un directorio predefinido (en UNIX normalmente es */usr/include*).

A continuación vamos a ver un ejemplo de un programa que utiliza la entrada/salida estándar; su nombre es **mayus**, y traduce su entrada a mayúsculas:

```
1 #include <stdio.h>
2 #include <ctype.h>
3 int main(void) /* convertir la entrada de minúsculas a mayúsculas */
4 {
5         int c;
6
7         while ((c = getchar()) != EOF)
8         putchar(islower(c) ? toupper(c) : c);
9 }
```

Comentemos algunas de las funciones utilizadas:

- **`getchar()`** devuelve el siguiente carácter de la entrada estándar cada vez que se le llama, y devuelve EOF cuando encuentra el fin de fichero en la entrada. La biblioteca estándar define la constante simbólica EOF como -1 (mediante un #define en stdio.h), pero las comparaciones deben realizarse en términos de EOF y no de -1 para hacerlas independientes del valor específico.
- **`putchar(c)`** envía el carácter `c` a la salida estándar.
- Las funciones **`islower`** y **`toupper`** son en realidad macros. La macro **`islower`** comprueba si un argumento es una letra minúscula, devolviendo un valor distinto de cero en caso afirmativo, o cero en caso contrario. La macro **`toupper`** convierte una letra minúscula en su mayúscula correspondiente.

Compilemos el programa y probemos su funcionamiento:

> **$ cc -o mayus mayus.c**
>
> **$ mayus**
>
> **Cogito ergo sum**
>
> COGITO ERGO SUM
>
> **ctrl-d**

El programa, efectivamente, transforma su entrada estándar a mayúsculas. La entrada, salida y error estándar (asignados por defecto al terminal) se pueden redireccionar a ficheros (y como hay ficheros que representan periféricos, también a periféricos) utilizando los metacaracteres de redireccionamiento. El carácter '>' seguido de una trayectoria de fichero sirve para redireccionar la salida estándar al fichero especificado tras '>'.

> **$ mayus >borrar**
>
> **Je pense donc je suis**

> **ctrl-d**
>
> **$ cat borrar**
>
> JE PENSE DONC JE SUIS

La orden **cat** seguida de un nombre de fichero de texto muestra en la salida estándar (el terminal por defecto) el contenido del fichero. El *shell* interpreta el metacarácter '>', realizando varias llamadas al sistema para que la salida estándar cambie del terminal al fichero *borrar*. El cambio de la salida estándar al fichero borrar es ajeno al programa **mayus**; en concreto la cadena *'>borrar'* no se incluye en los argumentos *argv* de la línea de órdenes. Para redireccionar la entrada estándar se utiliza el metacarácter '<' seguido de una trayectoria de fichero, esto hace que la entrada estándar del programa proceda de dicho fichero en lugar del terminal. Por ejemplo, creemos un fichero llamado *borrar2* que contiene una única línea con el texto "Pienso luego existo" y tecleemos

> **$ mayus <borrar2**
>
> PIENSO LUEGO EXISTO

Al igual que antes el *shell* es el encargado del redireccionamiento; en este caso se ha redireccionado la entrada estándar para que provenga del fichero *borrar2* en lugar del terminal (como antes, el programa **mayus** no percibe que su entrada ha sido redireccionada). Se puede redireccionar tanto la entrada como la salida estándar de una misma orden, como en *'mayus <borrar2 >borrar3'*. Existen muchas órdenes que leen y/o escriben en/de los ficheros estándares. Por ejemplo ls

> **$ ls**

genera una lista de nombres de ficheros en el terminal del usuario. Pero si se teclea

> **$ ls >listaficheros**

la misma lista de nombres de ficheros será colocada en el fichero *listaficheros*. El metacaracter '>' provoca que el fichero se cree en caso de no existir y, si ya existía, su contenido será reemplazado (hay que estar seguro al utilizar *'>fich'* de que si existe *fich* no nos interese su contenido). El símbolo '>>' opera igual que '>', excepto que aquel significa "anexa al final de". Esto es,

> **$ mayus <borrar2 >>borrar1**

pasa el contenido de *borrar2* a mayúsculas al final de lo que se encuentre en el fichero *borrar1*, en vez de reemplazar el contenido existente. Como en '>', si *borrar1* no existe se creará inicialmente vacío para el usuario.

En todos estos ejemplos, los espacios en blanco son opcionales a ambos lados de <, > o >>, si bien nuestro formato es el tradicional. Estos metacaracteres de redireccionamiento suelen coincidir en todos los *shells*, aunque usted es libre de escribir un *shell* que utilice otros caracteres para el redireccionamiento. El último metacarácter de redireccionamiento varía en los *shells* clásicos, su utilidad es redireccionar el error estándar, y suele expresarse como '>&'.

Existen multitud de programas en UNIX, como **mayus**, que leen de su entrada estándar, la transforman y escriben la entrada transformada en la salida estándar. A estos programas se les llama **filtros**. Su implementación se asemeja al siguiente programa, que es una nueva versión de **mayus** llamada **mayus2**:

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3 #include <stdlib.h>
 4 void mayus(FILE *fp);
 5 int main(int argc, char *argv[]) /* mayus2: convierte la entrada de minúsculas a mayúsculas */
 6
 7 {
 8         FILE *fp;
 9         if (argc == 1) /* ningún argumento, a mayúsculas la entrada estándar */
10         mayus(stdin);
11         else
12         if ((fp = fopen(argv[1], "r")) == NULL) {
13                 fprintf(stderr, "%s: no puedo abrir %s\n", argv[0], argv[1]);
14                 exit(1);
15         } else {
16                 mayus(fp);
17                 fclose(fp);
18         }
19 }
20
21 void mayus(FILE *fp) /* pasa a mayúsculas el fichero fp a la salida estándar */
22 {
23         int c;
24         while ((c = getc(fp)) != EOF)
25         putc(islower(c) ? toupper(c) : c, stdout);
26 }
```

Este programa merece algunos comentarios:

- Si el programa no recibe argumentos (argc == 1) transformará la entrada estándar a mayúsculas en la salida estándar. Si

recibe un argumento, que debe ser un fichero existente, lo pasa a mayúsculas igualmente en la salida estándar. Si no entiendes argc y argv consulta ¿cómo se pasan argumentos de entrada a una orden o programa?

- Cuando se quiere utilizar un fichero, es preciso realizar una operación de apertura para indicárselo al sistema operativo. Esto se realiza mediante la función *fopen* (que toma como argumentos la trayectoria del fichero y su modo de apertura), esta función nos devuelve un puntero a la estructura FILE definida en *stdio.h*. En las sucesivas llamadas al sistema relacionadas con el fichero se utilizará dicho puntero, en lugar de la trayectoria del fichero. Cuando no se desee utilizar más el fichero habrá que cerrarlo con la función *fclose* para que el sistema operativo libere los *buffers* y demás información que debe mantener por cada fichero abierto. Como se dijo anteriormente existen tres ficheros abiertos al iniciar un programa: la entrada, salida y error estándar. Sus punteros respectivos son constantes definidas en *stdio.h*, los nombres respectivos de dichas constantes son *stdin*, *stdout* y *stderr.*
- Se utiliza la función *getc* para leer el siguiente carácter de un fichero, esta función devuelve EOF cuando se alcanza el fin del fichero. *putc* es la inversa de *getc*, *putc(c, fp)* envía el carácter *c* al fichero *fp*, y devuelve *c*. *getchar* y *putchar* se puede definir en términos de *getc, putc*, *stdin* y *stdout* como sigue:

```
1 #define getchar() getc(stdin)
2 #define putchar(c) putc(c, stdout)
```

A continuación le sugerimos que experimente con mayus2, pruebe por ejemplo:

> **$ mayus2**
>
> **$ mayus2 <fich** *donde fich es el nombre de un fichero de texto*
>
> **$ mayus2 fich**
>
> **$ mayus2 no_existe** *no_existe es el nombre de un fichero que no existe*
>
> **$ mayus2 no_existe >&error** *se supone que >& redirecciona el error estándar*
>
> **$ cat error**

Comentemos la diferencia entre 'mayus2 <fich' y 'mayus2 fich'. Como se observa el resultado es el mismo, sin embargo, la cadena *<fich* es interpretada por el *shell*, **mayus2** no ve como argumento el nombre de fichero *fich*, sino que pasa a mayúsculas su entrada estándar, que el *shell* ha redireccionado para que provenga del fichero. Por el contrario, en la segunda orden se pasa el nombre *fich* como argumento para **mayus2**, el cual lee el fichero y lo pasa a mayúsculas.

Como se dijo, existen muchas órdenes en UNIX con la estructura de **mayus2**, un ejemplo es **sort** (que como su nombre indica sirve para ordenar líneas) cuya sintaxis es:

sort [opciones] [lista de ficheros]

es posible pasarle a **sort** una lista de nombres de fichero, como en

> **$ sort fich1 fich2 fich3**

pero si no se le pasan argumentos también se puede utilizar para ordenar la entrada estándar:

> **$ sort**
>
> **ghi**
>
> **abc**
>
> **def**
>
> **ctrl-d**
>
> abc
>
> def
>
> ghi

Otra orden muy útil es **cat**, utilizada para concatenar (*conCAT*) ficheros de texto, su sintaxis es:

$ cat [opciones] [lista de ficheros]

cat concatena los ficheros y saca dicha concatenación por la salida estándar, algunas variantes de uso de **cat** muy utilizadas son:

> **$ cat >nue_fich** *se utiliza para crear pequeños ficheros de texto sin utilizar un editor*
>
> **$ cat fich** *muestra el contenido de fich*

> **$ cat fich1 fich2 ... >>x** *anexa al final del fichero x los contenidos de fich1 fich2 ...*

5.2.3 Interconexiones (pipes)

Muchas veces es útil combinar la acción de varias órdenes. Por ejemplo, la orden **who** muestra en la salida estándar una línea con información sobre las sesiones abiertas en el sistema. La primera información de dicha línea es el nombre del usuario que tiene abierta la sesión. Se puede combinar la salida de esta orden con **sort** para obtener una lista ordenada por nombre de usuario de las sesiones abiertas:

> **$ who >temp**
>
> **$ sort <temp**

Como este tipo de combinaciones son muy utilizadas UNIX proporciona una llamada al sistema para redireccionar la entrada estándar de un proceso para que en vez de provenir del terminal provenga de la salida estándar de otro proceso. Los *shells* hacen uso de dicha llamada al sistema, y proporcionan un metacarácter, el '|', para poder especificar la redirección. Así, si escribimos:

> **$ who | sort**

el *shell* creará dos procesos, y conectará la salida estándar de **who** a la entrada estándar de **sort**. La orden **who** no es consciente de que su salida estándar ha sido redireccionada, ni la orden **sort** de que lo ha sido su entrada estándar. Entre los dos procesos se ha creado un canal de comunicación llamado *pipe*, esta palabra, de origen inglés, ha sido traducida principalmente como **interconexión**, **tubo** y **tubería**.

Las interconexiones sirven para combinar la acción de varias órdenes (aprovechando que la mayoría de las órdenes de UNIX utilizan su entrada y/o salida estándar) creando órdenes más potentes, he aquí varios ejemplos:

> **$ ls | wc -l**
>
> **$ who | grep pepe**
>
> **$ who | grep pepe |wc -l**

la orden **wc** (*Word Counter*) sirve para contar el número de caracteres, palabras y líneas de la entrada estándar, o de uno o varios ficheros de texto que se pasen como argumentos. Con la opción **-l** sólo cuenta el número de líneas. Como **ls** lista una línea por fichero o directorio, **ls | wc -l** mostrará el número de ficheros y directorios que cuelgan del directorio de trabajo. La orden **grep** selecciona de su entrada estándar (o de los ficheros dados como argumentos) las líneas que contienen un patrón de texto (en el ejemplo *pepe*), por lo tanto la última interconexión nos va a servir para contar el número de sesiones abiertas por el usuario *pepe*.

Por último, hay que decir que se pueden combinar en una misma orden interconexiones y redireccionamientos, como en:

> **$ ls | wc -l >num**

6 Ejercicios propuestos

1) Realice un programa en C equivalente a la orden **echo**, es decir, un programa que lo único que haga sea mostrar sus argumentos. Llámelo **eco**. Ejecute

> **$ eco a b**
>
> **$ eco ***

¿por qué en la segunda línea eco no muestra como salida un asterisco? Pruebe a realizar **$ echo ***. ¿Intuye para qué sirve la barra invertida?

2) Dada la orden

> **$ cat a b >x**

¿qué valores tendrán las variables *argc* y *argv* para dicha ejecución del programa **cat**? (Nota: lo puede deducir empleando la orden **eco**, propuesta en el ejercicio anterior).

3) Suponga que en su directorio de trabajo no existe el fichero *usuarios*. Escriba mal el nombre de una orden, como en

> **$ woh >usuarios**
>
> woh: Command not found

y compruebe mediante la orden **ls** si existe el fichero *usuarios*. ¿Por qué cree que existe? ¿Por qué tiene tamaño 0?

4) Dadas las órdenes **head** y **tail**, con la siguiente sintaxis y semántica:

head -n [*fich*] muestra en la salida estándar las *n* primeras líneas de la entrada estándar, o del fichero *fich*, si se usa segundo parámetro.

tail -n [*fich*] muestra en la salida estándar las *n* últimas líneas de la entrada estándar, o del fichero *fich*, si se usa segundo parámetro.

escriba una orden que muestre en la pantalla la tercera línea del fichero */etc/passwd*.

Anexo A: Glosario

¿Cómo se pasan argumentos a una orden?

Habitualmente, cuando se lanza un programa a ejecución desde el shell, se le añaden parámetros o argumentos para definir exactamente qué queremos de él. Por ejemplo:

```
vi pepe.txt

ls -l /usr/include

cp pepe.c ../pipo.c
```

En el primer caso se invoca al editor `vi` especificándose que se desea trabajar con el fichero "`pepe.txt`". El fichero es un parámetro pasado al shell. El segundo caso es una llamada al programa ls que incorpora dos parámetros, como ocurre en el último ejemplo.

¿Qué son los parámetros o argumentos? En principio, puede afirmarse que son conjuntos de caracteres separados por blancos (espacios o tabuladores). Ahora bien, no se consideran parámetros los redireccionamientos, y caracteres como el ampersand "&" o el punto y coma "`;`" actúan de separadores (en general, eso ocurre con todos los caracteres que tengan un significado para el intérprete de órdenes).

> Ejemplo: En la orden
>
> `ls -l /usr/include & >pepe.txt`
>
> los parámetros son "`ls`", "`-l`" y "`/usr/include`" ('&' y la redirección no cuentan).

Cada programa recibe los parámetros a través de su punto de entrada, que en el caso del lenguaje C es la función **main**. El formato mínimo que acepta esta función en UNIX/Linux es:

```
main ( int argc, char* argv[] );
```

donde **argc** expresa cuántos parámetros se han reconocido, y **argv** es un vector de cadenas de caracteres que precisamente contienen los parámetros, siendo `argv[i]` el parámetro `i`.

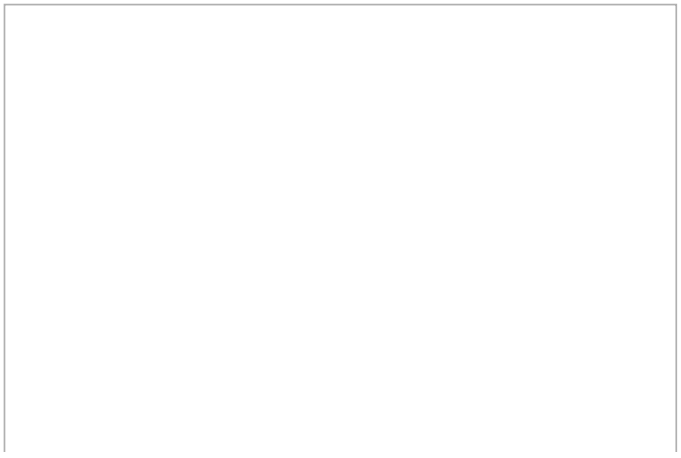
Los parámetros se empiezan a numerar en 0. El parámetro 0 es el nombre del programa invocado tal y como se pasó en la línea de órdenes. En el ejemplo de `vi pepe.txt`, los valores de **argc** y **argv** serían:

```
argc = 2

argv[0] = "vi"

argv[1] = "pepe.txt"
```

¿Para qué sirven las llamadas fork y wait?



Para crear nuevos procesos, el UNIX/Linux dispone únicamente de una llamada al sistema, **fork**, sin ningún tipo de parámetros. Su prototipo es

```
int fork();
```

Al llamar a esta función *se crea un nuevo proceso (proceso hijo), idéntico en todo (en código, datos y Bloque de Control de Proceso) al proceso que ha realizado la llamada (proceso padre) pero en una nueva zona de memoria*. Los espacios de memoria del padre y el hijo son disjuntos, por lo que el proceso hijo es una copia idéntica del padre que a partir de ese momento sigue su vida separada, sin afectar a la memoria del padre; y viceversa.

Siendo más concretos, las variables del proceso padre son inicialmente las mismas que las del hijo. Pero si cualquiera de los dos procesos altera una variable, el cambio sólo