Introducción a Sistemas Operativos: Empezando

Francisco J Ballesteros

1. ¿Qué es el Sistema Operativo?

El sistema operativo es software que te permite utilizar el ordenador. Lo que esto signifique dependerá del punto de vista del usuario que utiliza el ordenador. Por ejemplo, para mi abuela el sistema operativo incluye no sólo Windows, sino también todos los programas instalados en el ordenador. Para un programador, la mayoría de las aplicaciones no forman parte del sistema operativo. No obstante, este usuario podría considerar los compiladores librerías y utilidades para programar como parte del sistema. Para un programador de sistemas, los programas que forman el sistema operativo serían muchos menos de los que otros usuarios considerarían como parte del sistema. Todo depende del punto de vista.

Este curso intenta enseñar cómo utilizar el sistema operativo de forma efectiva y qué abstracciones forman parte del mismo. En adelante, nos referimos al sistema operativo como *sistema*, para abreviar. Usar el sistema implica utilizar las funciones que incluye y los programas y lenguajes que forman parte del mismo y nos permiten utilizar el ordenador. El propósito es siempre el mismo: Hacer que la máquina haga el trabajo y evitar tener que hacerlo manualmente. La diferencia entre saber utilizar el sistema y no saber hacerlo es la diferencia entre necesitar horas o días para hacer el trabajo y necesitar un par de minutos para conseguirlo. La elección es tuya, aunque parece clara.

En este curso aprenderás a utilizar un sistema, a ver qué hace y que abstracciones proporciona y a tener una idea aproximada de cómo lo hace. Tienes otros libros que cubren otros aspectos:

- Para aprender C siempre tienes [1].
- Como libro de introducción a UNIX tienes [2].
- Puedes ver [3] para aprender conceptos teóricos de sistemas operativos y su relación con la implementación.
- El mejor libro para ver cómo está hecho el sistema es quizá [4]. Aunque describe la 6th edición de UNIX, sigue siendo el mejor. Una vez digieras este libro, podrías seguir con [5] y leer el código fuente de OpenBSD mientras lo lees.
- Para continuar aprendiendo a programar sobre UNIX cuando termines este curso puedes utilizar [6].
- En [7] tienes consejos útiles sobre como programar.
- Como referencia (para buscar funciones y programas) siempre tienes el manual en línea en cualquier sistema UNIX (y en internet los tienes todos).

Pero volvamos nuestra pregunta... ¿Qué es el sistema operativo? Es tan sólo un conjunto de programas que te permiten utilizar el ordenador. El hardware es complejo y está lejos de los conceptos que utilizas como programador (no digamos ya como usuario). Hay multitud de tipos de procesadores, dispositivos hardware para entrada/salida (o E/S, o I/O, por *input/output*), y muchos otros artefactos. Si tuvieras que escribir el software necesario para manejar todos los que usas, no tendrías tiempo de escribir el software de la aplicación que quieres escribir. El concepto es pues similar al de una librería (o biblioteca) de software. De hecho, los sistemas operativos empezaron como librerías utilizadas por aquellos que escribían programas para una máquina.

Cuando el ordenador arranca, el procesador comienza a ejecutar instrucciones de un programa que

habitualmente se guarda en memoria no volátil. Este programa es un cargador cuyo trabajo es localizar en el disco o en algún otro dispositivo el código de otro programa, el núcleo del sistema operativo, y cargarlo en la memoria. Una vez cargado, se salta a su primera instrucción y lo que suceda a partir de ese momento depende del sistema operativo que estemos ejecutando. Se suele llamar *kernel* al núcleo del sistema operativo. Es un programa como cualquier otro programa, pero es importa puesto que permite que los programas de los usuarios puedan ejecutar y, por ello, permite a los usuarios utilizar el ordenador. Tener un sistema operativo tiene tres ventajas:

1. No es preciso escribir el código que incluye el sistema operativo, ya lo tenemos escrito y lo podemos utilizar sea cual sea la aplicación que queremos ejecutar.
2. Podemos olvidarnos de los detalles necesarios para utilizar el hardware. El sistema operativo se comporta como nuestra librería y ya incluye tipos abstractos de datos que empaquetan los servicios que nos da el hardware de un modo más apropiado.
3. Podemos olvidarnos de cómo se pueden repartir los recursos del hardware entre los distintos programas que queremos ejecutar en el mismo ordenador. El sistema operativo está hecho para que sea posible utilizarlo de forma simultánea desde varios programas en el mismo equipo.

La mayoría de los programas que has escrito utilizan discos, pantallas, teclados y otros dispositivos. No obstante, los has podido escribir sin saber cómo se manipulan. Dicho de otro modo, no has tenido que escribir el software que sabe como manejarlos (no has tenido que escribir los *manejadores* o *drivers* para ellos). Esto debería ser razón suficiente para convencerte de la utilidad del sistema operativo.

Pero hay más. Los tipos abstractos de datos son muy cómodos para escribir software. De hecho, son casi indispensables. Por ejemplo, has escrito programas que utilizan ficheros. No obstante, los discos donde guardas tus ficheros no saben nada respecto a ellos: ¡El hardware no sabe lo que es un fichero! El disco sólo sabe cómo almacenar bloques de bytes. Lo que es más, sólo sabe como almacenar bloques del mismo tamaño (normalmente llamados *sectores*). A pesar de ello, nosotros preferimos utilizar nombres para cada conjunto de datos de nuestro interés. Y preferimos que dichos datos persistan, almacenados en el disco. Nos los imaginamos con una serie contigua de bytes en el disco, empaquetados dentro de un "fichero". Es el sistema operativo el que se inventa el tipo de datos *fichero* y suministra las operaciones que permiten utilizarlo. Incluso el nombre de un fichero es parte de la abstracción. Es otra "mentira" implementada por el sistema operativo.

Esto es tan importante que incluso el hardware lo hace. Volviendo a pensar en un disco, el interfaz que utiliza el sistema operativo es habitualmente un conjunto de registros que permiten leer bloques del disco y escribir bloques en el. El sistema piensa que el disco es una sucesión (un *array*) de bloques contiguos, identificados por una dirección (el índice en el array). Todo esto es mentira. En el hardware de la tarjeta que controla el disco hay gran cantidad de software que está ejecutando e inventando esta abstracción (array de bloques). En la actualidad, sólo los que trabajan para un fabricante de disco saben realmente lo que hace el disco internamente. Todo lo demás son abstracciones implementadas por el software que, en este caso, podríamos decir que es el sistema operativo del disco, o el *firmware* del disco. Los discos pueden tener geometrías complejas para optimizar el acceso y pueden contener caches que mejoren su rendimiento. En cualquier caso, el sistema operativo piensa que un disco es básicamente un array de bloques. Exactamente lo mismo sucede cuando tus programas utilizan ficheros.

Utilizar tipos abstractos de datos tiene otra ventaja: La portabilidad. Si el hardware cambia, pero el tipo de datos que utilizas sigue siendo el mismo, no es preciso que cambies el programa. Tu programa seguirá funcionando. ¿Has visto que tus programas utilizan ficheros sin pensar en qué tipo de disco se utiliza para almacenarlos? Los ficheros se utilizan igual tanto si son ficheros almacenados en un disco magnético, en un disco USB o en cualquier otro medio de almacenamiento.

Piensa que el hardware puede cambiar también cuando lo reemplazas por hardware más moderno. Los sistemas operativos están hechos de tal forma que sea posible utilizar versiones antiguas del hardware. Dicho de otro modo, suelen tener *compatibilidad hacia atrás*. Esto quiere decir que están hechos intentando que los programas sigan funcionando a pesar de que el hardware evolucione. Simplemente, en algún momento

tendrás que actualizar el sistema instalando nuevos programas para el nuevo hardware (sus *drivers*). Tus programas seguirán ejecutando del mismo modo.

Por esta razón decimos que el sistema operativo es una *máquina virtual*, que corresponde a una máquina que no existe. Por eso se la denomina "virtual". La máquina suministra ficheros, procesos (programas en ejecución), ventanas, conexiones de red, etc. Todos estos artefactos son desconocidos para el hardware.

Dado que los ordenadores son extremadamente rápidos, es posible utilizarlos para ejecutar varios programas de forma simultánea. El sistema hace que sea sencillo mantener todos los programas ejecutando a la vez (o casi a la vez). ¿Has notado que resulta natural programar pensando que el programa resultante tendrá toda la máquina para el solo? No obstante, ese no será el caso. Es casi seguro que tendrás ejecutando al menos un editor, un navegador web y otros muchos programas. El sistema decide qué partes de la máquina, y en qué momentos, se dedican a cada programa. Esto es, el sistema *reparte*, o *multiplexa*, los recursos del ordenador entre los programas que lo utilizan. Las abstracciones que suministra el sistema tratan también de aislar unos programas de otros, de tal forma que sea posible escribir programas sin necesidad de pensar en todo lo que tenemos ejecutando dentro del ordenador.

Por esto decimos que el sistema operativo es un *gestor de recursos*, o un *multiplexor* de recursos. Asigna recursos a los programas y los reparte (o multiplexa) entre ellos. Algunos recursos, como la memoria, los podemos repartir dando a cada programa un trozo del recurso: los multiplexamos en el espacio. Unos programas utilizan unas partes de la memoria, y otros utilizan otras. El kernel también necesita utilizar su propia parte. Otros recursos, como el procesador, los tenemos que repartir dando a cada programa el recurso entero durante un tiempo. Pasado ese tiempo, el recurso se dedica a otro programa. Estos recursos se multiplexan en el tiempo. Dado que la máquina es tan rápida, da la impresión de que todos los programas están ejecutando a la vez (en paralelo). No obstante, si tenemos un único procesador (un *core*), sólo podemos ejecutar un programa en cada instante. Pero incluso en este caso, todos los programas ejecutan *concurrentemente* (de forma simultánea).

La última misión del sistema operativo tiene que ver con esto. Los humanos y los programas cometen errores. Además, los programas tienen *bugs* (que no son otra cosa que errores cometidos por sus programadores). Un error en un programa puede hacer que todo el ordenador se venga abajo y deje de funcionar, a menos que el sistema operativo tome medidas para evitar esto. No obstante, el sistema operativo no es una divinidad y tan sólo puede utilizar los mecanismos de protección que suministra el hardware para intentar proteger a unos programas de otros. Por ejemplo, una de las primeras cosas que hace el kernel cuando se inicializa es proteger su propia memoria. Esta se marca como privilegiada y se permite el acceso a la misma sólo para código que ejecute con el procesador en "modo privilegiado" (con un bit en un registro puesto a un valor determinado). El kernel ejecuta en modo privilegiado, pero tus programas no lo hacen. Al saltar hacia el código del usuario, el procesador de deja en modo no privilegiado y el efecto neto es que tus programas no pueden acceder a la memoria del kernel. Además, las protecciones de la memoria que utiliza un programa se ajustan de tal modo que otros programas no puedan acceder a la misma. Como resultado, un error en un programa no conseguirá habitualmente que otros programas dejen de funcionar. ¿Has notado que cuando tu programa tiene un bug, otros programas pueden seguir funcionando? ¿Podrías decir ahora por qué?

Resumiendo, el sistema operativo es un programa que te da abstracciones para utilizar el hardware, que te permiten programar de un modo mas simple. Y naturalmente, reparte dicho hardware entre los programas que lo usan. Para hacerlo, ha de gestionar los recursos y proteger a unos programas de otros. En cualquier caso, el sistema es tan sólo un programa.

2. ¿Qué es UNIX?

En este curso vamos a utilizar UNIX como sistema operativo. Hoy en día, eso quiere decir Linux, OpenBSD (u otros cuyo nombre termina en "BSD") o MacOS (OS X).

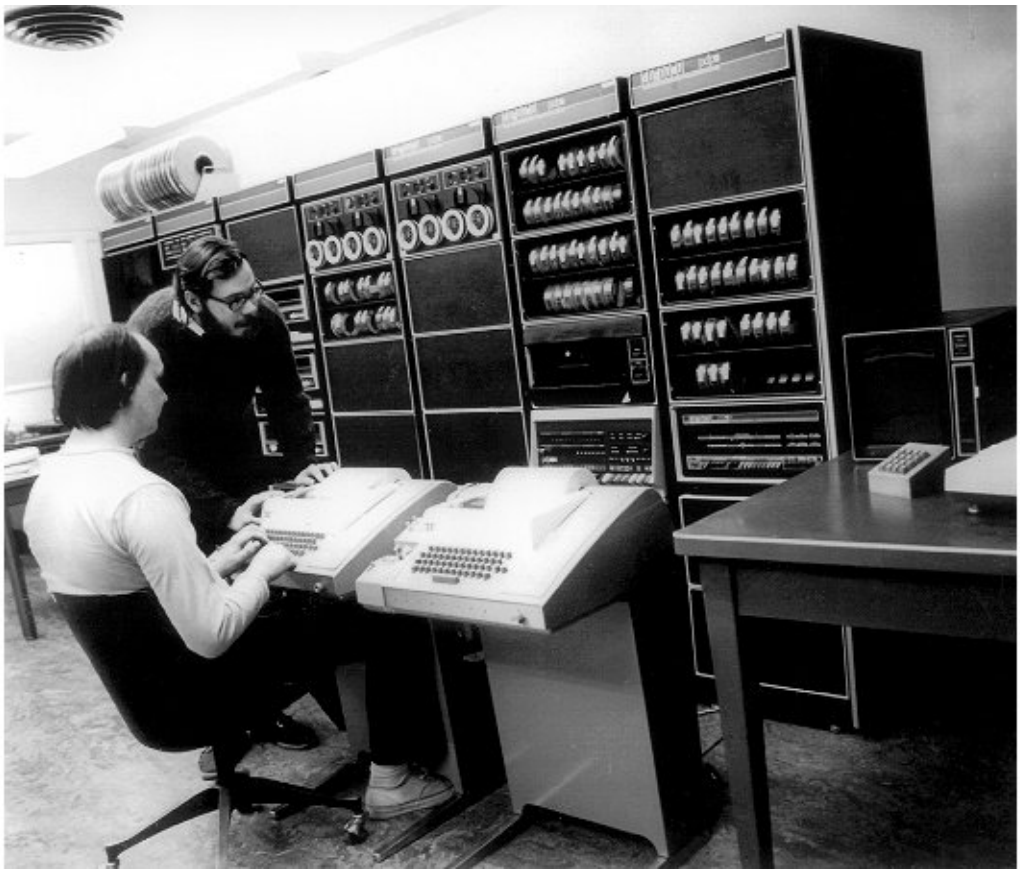

Figura 1: *Ken Thompson y Dennis Ritchie junto a la PDP. ¡Seguro que lo pasaban bien!*

Hace mucho tiempo (pero en esta galaxia) Ken Thompson y Dennis Ritchie hicieron un programa llamado UNIX para un ordenador llamado PDP de una empresa, Digital, que ya no existe. El sistema era tan fácil de utilizar y de adaptar para otros ordenadores que se extendió como la pólvora. Como su código fuente era fácil de entender (en comparación con los de otros) hay muchos que lo han modificado y, como resultado, hay toda una familia de sistemas operativos que descienden de UNIX (y se conocen como UNIX de hecho) pero que, naturalmente, difieren en alguna medida. Así pues, UNIX realmente no existe, aunque es el sistema más popular hoy día para programar y para mantener servicios en red. Consulta los libros que hemos mencionado para ver más sobre esto.

Oirás o leerás nombres como *POSIX* (*Portable Operating System Interface*) y como X/Open o SUS (*Single Unix Specification*). Estos se refieren a diversos estándares que intentan estandarizar (¡sorprendentemente!) UNIX. La práctica totalidad de los sistemas UNIX difieren de dichos estándares y, si se utilizan opciones para hacer que no lo hagan, en la mayoría de los casos encuentras problemas. En pocas palabras, puedes suponer que dichos nombres se refieren también a UNIX, aunque a un UNIX que no existe y que es una amalgama del comportamiento de los que existían cuando se escribieron dichos estándares.

3. ¿Cómo seguir este curso?

Para seguir este curso lo primero que has de hacer es buscar un sistema UNIX e instalarlo. Quizá ya lo tienes si utilizas Linux o un Mac. En caso contrario es muy fácil descargar Linux e instalarlo (sugerimos Ubuntu por su facilidad de instalación, no es que nos guste más). Conforme avances leyendo o atendiendo a las clases, utiliza tu UNIX para probar los programas, llamadas y comandos que vamos viendo. ¡Juega con ellos! Pocas cosas te resultarán tan rentables profesionalmente (¡Y económicamente!) como aprender **bien** a utilizar UNIX.

Cuando termines este curso podrás programar y utilizar cualquiera de ellos, ¡Y otros muchos!. Piensa que sistemas como Windows y otros que no son UNIX utilizan básicamente las mismas abstracciones y, con el tiempo, han ido incorporando las ideas de UNIX.

Si deseas que tu programa o los comandos que utilizas sean **portables**, se puedan utilizar en distintos sistemas UNIX, un buen consejo es que mantengas instalado al menos un sistema Linux y un OpenBSD y te asegures de que tus programas funcionan en ambos. Cuando lo hagan, es muy posible que funcionen correctamente en otros sistemas UNIX. En cualquier caso, las páginas de manual te informan de si las llamadas y comandos son específicos del sistema que usas o forman parte de algún estándar.

4. Llamadas al sistema

¿Qué relación tiene el sistema operativo con tus programas? ¿Cómo entender lo que ocurre cuando ejecutas un programa en el sistema operativo? Verás que las cosas son más simples de lo que parecen. Para entenderlo, vamos a escribir un pequeño programa en el lenguaje de programación C. Por el momento, puedes ignorar cómo se escribe el texto del programa (el código fuente) y qué comandos son precisos para ello. El código para un programa que escribe hola podría ser el que sigue:

`[hola.c]`:
```
int
main(int argc, char *argv[])
{
    puts("hola");
}
```

Este texto, escrito con un editor de texto, podría estar guardado en un fichero llamado `hola.c` (en todos los listados incluimos el nombre del fichero debajo del listado). Naturalmente, el ordenador no sabe cómo ejecutar este fichero. Hemos de traducirlo a un lenguaje que entienda el procesador, a código binario. Para ello, podemos utilizar otro programa, denominado compilador. El compilador lee el fichero con el código que hemos escrito, llamado código fuente, y lo traduce a datos que son suficientes para cargar un binario en la memoria del ordenador. En realidad, primero se genera un fichero con código objeto que contiene el binario para el fuente contenido en el fichero que compilamos:

```
unix$ cc -c hola.c
```

Hemos utilizado el comando *cc* para compilar ("C compiler") y obtener un fichero `hola.o`. El texto "`unix$`" es el *prompt* que escribe nuestro intérprete de comandos, o shell, para indicar que podemos escribir comandos. Y ahora podemos enlazar dicho fichero objeto para obtener un fichero ejecutable:

```
unix$ cc hola.o
```

Hemos utilizado *cc* también para enlazar. El resultado será un fichero llamado `a.out` con el ejecutable. Ahora podemos ejecutarlo utilizando su nombre:

```
unix$ a.out
hola
```

Para ejecutarlo, el sistema operativo se ha ocupado de cargar el binario en la memoria y de saltar a la primera instrucción del mismo.

Podemos utilizar el comando *ls* para listar los ficheros que hemos utilizado:

```
unix$ ls -l
total 64
-rwxr-xr-x  1 elf  wheel  8570 May  4 16:03 a.out
-rw-r--r--  1 elf  wheel    75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel  1288 May  4 16:03 hola.o
```

Ignorando cosas que no nos interesan, el comando *ls* nos informa de que el fichero `hola.c` contiene 75 bytes, el fichero `hola.o` contiene 1288 bytes y el ejecutable `a.out` contiene 8570 bytes.

El ejecutable contiene no sólo el código objeto (binario) para el fuente que hemos escrito. Contiene también código para funciones que utilizamos y que, en este caso, proceden de la librería de C. En nuestro programa llamamos a *puts*, que es una función de C que llamará a otra función, *write*, para escribir el texto que le hemos indicado. El aspecto puede verse en la figura 2.

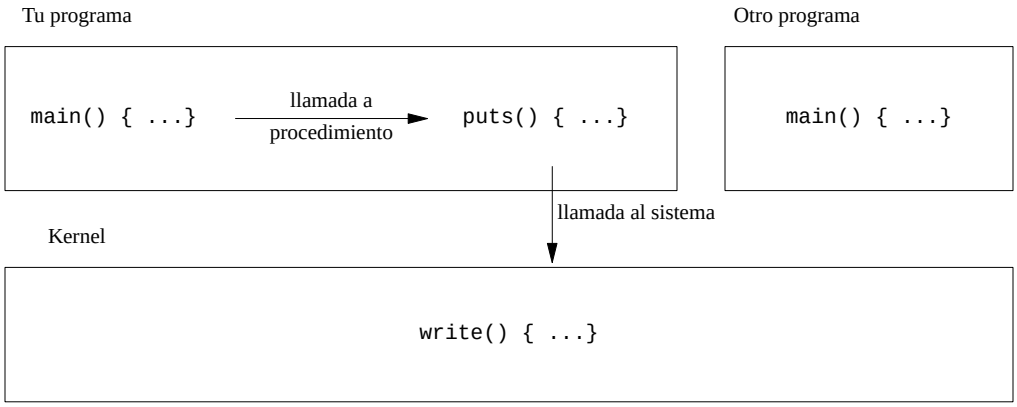

Figura 2: *Una llamada al sistema, utilizada por nuestro programa para saludar.*

Cuando el programa ejecuta empieza en *main*. Esta función llama a otra función, *puts*, que está implementada en la librería de C, y se ha enlazado junto con nuestro código objeto para formar un ejecutable. Pues bien, *puts* está implementada con una llamada a otra función, *write*, que no está en realidad dentro del ejecutable. En realidad, sí que hay una función *write*, pero está prácticamente vacía. Lo único que hace es dejar en los registros alguna indicación (un número en un registro, por ejemplo) de la llamada que se quiere hacer a UNIX y utilizar una instrucción para provocar la llamada. El efecto de esta instrucción es similar al de una interrupción. Hace que el hardware transfiera el control a una dirección determinada dentro del kernel del sistema operativo. Dicha llamada, *write* en este caso, ejecuta y (una vez completa) retornará el control como si de una llamada a procedimiento se tratase. En realidad, se utiliza otra instrucción para retornar de nuevo al código que llamó al sistema y su efecto es similar a retornar de una interrupción: el hardware recupera el valor de los registros que salvó al entrar al sistema, y se continúa ejecutando en la instrucción siguiente a la llamada.

Como puedes ver, el kernel se comporta en realidad como una librería. Normalmente no hará nada hasta que un programa de usuario haga una llamada al sistema. Naturalmente, parte del trabajo del kernel es atender las interrupciones que proceden del hardware. Por ejemplo, al pulsar una tecla en el teclado provocas una interrupción. En ese momento, el hardware salva el estado de los registros y salta a una dirección en la memoria para atender la interrupción. Dicha dirección ha sido indicada por el sistema operativo durante su inicialización. En nuestro ejemplo, será la dirección del programa que maneja la interrupción de teclado, que está dentro del kernel. Una vez atendida la interrupción, el kernel retorna de la misma y el programa de usuario que estaba ejecutando continúa una vez el hardware restaure en el procesador los registros que había salvado al iniciar la interrupción. Para el programa de usuario nunca ha sucedido nada. En cualquier caso, podemos pensar en las interrupciones como llamadas al kernel procedentes del hardware, con lo que vemos que el kernel sigue siendo una librería a todos los efectos.

Anteriormente dijimos que parte del trabajo del sistema operativo era inventar tipos abstractos de datos para ficheros, programas en ejecución, conexiones de red, etc. Las operaciones correspondientes a cada una de esas abstracciones son las *llamadas al sistema* que ha de implementar el kernel. ¡No es muy diferente a cuando inventas un tipo de datos en un programa y programas operaciones para manipular variables de dicho tipo!

5. Comandos, Shell y llamadas al sistema.

En el epígrafe anterior hemos utilizado comandos para compilar nuestro programa y ejecutarlo. Vamos a ver qué es eso. En general, el único modo de utilizar un sistema es ejecutar un programa que haga llamadas al sistema. En nuestro ejemplo anterior, el programa *hola* (el ejecutable) ha llamado a *puts*. Esta función ha llamado a *write*, que es una llamada al sistema que en nuestro caso ha escrito en la pantalla.

Cuando el kernel se inicializa, ejecuta un programa sin que nadie se lo pida. El propósito de dicho programa es ejecutar todos los programas necesarios para que un usuario pueda utilizar el sistema. En UNIX, habitualmente se llama *init* a tal programa. En muchos casos, *init* localiza los terminales (pantallas y teclados) disponibles para los usuarios y ejecuta otro programa en cada uno de ellos.

Nosotros denominaremos *login* al programa que se ocupa de dar la bienvenida a un usuario en un terminal. En realidad, hoy día, es más habitual que dicho programa sea un programa gráfico que permita utilizar un teclado, un ratón y una pantalla para escribir un nombre de usuario y una contraseña o *password*. Esto es un ejemplo en un terminal con sólo texto (sin gráficos):

```
login: nemo
```

El programa *login* ha escrito "`login:`" y nosotros hemos escrito un nombre de usuario ("`nemo`" en este caso). Una vez pulsamos *enter* en el teclado, *login* lee el nombre de usuario y nos solicita una contraseña:

```
login: nemo
password: ******
```

Tras comprobar (¡Utilizando llamadas al sistema!) que el usuario es quién dice ser, esto es, que la contraseña es correcta, *login* ejecuta otro programa que permite utilizar el teclado para escribir comandos. A este programa lo llamamos *intérprete de comandos*, o *shell*. Continuando con nuestro ejemplo, tras pulsar de nuevo *enter* tras la escribir la contraseña, podríamos ver algo como sigue:

```
login: nemo
password: ******
Last login: Wed May  4 16:44:33 on ttys001
unix$
```

Las últimas dos líneas las ha escrito UNIX (*login* y el shell) y nos indican que hemos iniciado una sesión en el sistema. El texto "`unix$`" lo escribe el shell para indicar que está dispuesto a aceptar comandos. Lo denominamos *prompt*.

En este punto, es posible escribir una línea de texto y pulsar enter. Dicha línea es una **línea de comandos**. El shell lee líneas de comandos desde el teclado, y ejecuta programas para llevar a efecto los comandos que indicamos. En realidad, cuando pulsamos teclas...

1. El teclado envía interrupciones
2. El kernel las atiende y guarda el carácter correspondiente a cada tecla
3. Al pulsar *enter*, el kernel le da el texto que ha guardado al programa que esté "leyendo" de teclado.

Podemos por ejemplo ejecutar el comando que nos dice qué usuario somos:

```
unix$ who am i
nemo     ttys001  May  4 16:44
unix$
```

De nuevo, "`unix$`" es el prompt, y no lo hemos escrito nosotros. El shell ha leído la cadena de texto "`who am i`" y ha dividido dicha línea en palabras separadas por blancos. La primera palabra es "`who`", lo que indica que queremos ejecutar un comando denominado *who*. El resto de palabras se denominan **argumentos** y, en este caso, queremos utilizar como argumentos "`am`" y también "`i`". Así pués, el shell ha localizado un programa ejecutable (guardado en un fichero) con nombre "`who`" y le ha pedido a UNIX que lo ejecute.

Esto lo ha hecho utilizando llamadas al sistema. Además, le ha pedido a UNIX que le indique que sus argumentos son cada una de las palabras que hemos escrito.

En este punto el shell espera a que el comando termine y, mientras tanto, el programa que implementa este comando ejecuta y utiliza llamadas al sistema para hacer su trabajo. En este caso, informar que nuestro usuario es "`nemo`" y de qué terminal estamos usando.

Una vez el comando termina de ejecutar, UNIX avisa al shell de que tal cosa ha sucedido y, a continuación, el shell escribe de nuevo el prompt y lee la siguiente línea.

Podríamos ejecutar otro comando para ver qué usuarios están utilizando el sistema:

```
unix$ who
nemo     console  Jul 13 07:30
nemo     ttys000  Jul 13 07:31
nemo     ttys001  Aug 18 15:59
unix$
```

En este caso hemos usado el shell de nuevo para ejecutar `who`. Esta vez, sin argumentos. Por ello `who` entiende que se desea que liste qué usuarios están utilizando el sistema.

Los primeros argumentos de muchos comandos pueden comenzar con un "`-`" para activar **opciones** o flags que hagan que varíen su comportamiento. Por ejemplo,

```
unix$ uname
OpenBSD
unix$
```

imprime el nombre del sistema en que estamos, pero *con la opción* "`-a`"

```
crun% uname -a
OpenBSD crun.lsub.org 5.7 LSUB.MP#3 amd64
unix$
```

(que significa "*all*") imprime además el nombre de la máquina, la versión del sistema, el nombre de la configuración del kernel que estamos utilizando y la arquitectura de la máquina. Las opciones son argumentos pero el convenio es que comienzan por "`-`" y se indican al principio.

Como puedes ver, todo son llamadas al sistema en realidad. Lo único que sucede es que muchas veces las realizan programas que ejecutan debido a que ordenamos al sistema que los ejecute, debido a que ejecutamos comandos. Cuando utilizas ventanas, todo sigue siendo igual. El programa que implementa las ventanas (las dibuja y hace creer que funcionan) se llama *sistema de ventanas* y en realidad es otro *shell*. Sencillamente te permite utilizar el ratón para ejecutar comandos, además de escribirlos en ventanas de texto que ejecuten un shell tradicional.

Si ahora vuelves a leer los comandos que ejecutamos para compilar nuestro programa en C, todo debería verse más claro.

6. Obteniendo ayuda

La mayoría de los sistemas UNIX incluyen el manual en línea. Esto es, disponemos de comandos para acceder al manual. Saber utilizar el manual en UNIX es equivalente a saber utilizar Google en internet.

El manual se divide en secciones. Cada sección tiene una serie de páginas, cada una dedicada al elemento que documenta:

- La sección 1 está dedicada a comandos. Esto quiere decir que cada página de manual de la sección 1 documenta un comando (o varios).

- La sección 2 está dedicada a llamadas al sistema. Cada página de la sección 2 documenta una o varias llamadas al sistema.
- La sección 3 está dedicada a funciones de la librería de C que podemos utilizar para programar en dicho lenguaje. De nuevo, cada página documenta una o varias funciones.
- La sección 8 documenta programas dedicados a la administración del sistema (por ejemplo, formateo o inicialización de discos, etc.).

Dependiendo del UNIX que utilizamos, el resto de secciones suelen variar. Para aprender a utilizar el manual podemos utilizar el comando *man* y pedirle la página de manual de *man* en la sección 1:

```
unix$ man 1 man
```

O simplemente

```
unix$ man man
```

Eso produciría un resultado similar al que sigue.

```
unix$ man man
man(1)                                   man(1)

NAME
       man  -  format  and  display the on-line
       manual pages

SYNOPSIS
       man [-acdfFhkKtwW] [--path] [-m  system]
       [-S section_list] [section] name ...

DESCRIPTION
       man formats  and  displays  the  on-line
       manual  pages.   If you specify section,
       man only looks in that  section  of  the
       manual.   name  is  normally the name of
...
```

En la mayoría de los casos, el texto de la página de manual no cabe en tu pantalla y tendrás que pulsar el espacio en el teclado para avanzar. Pulsando la tecla "q" (quit) puedes abandonar la página y pulsando "b" (backward) puedes retroceder. Tu manual documenta como navegar por el texto utilizando el teclado.

Si no indicamos qué sección del manual nos interesa y existen páginas en varias secciones con el nombre que hemos indicado, *man* nos mostrará una de ellas (o todas ellas, dependiendo del UNIX que utilicemos).

La primera línea de una página de manual describe el nombre de la página (*man*) y la sección en que se encuentra (1 en nuestro caso). Habitualmente escribimos "ls(1)" para referirnos a la página de manual *ls* en la sección 1. Normalmente, sigue una sección "NAME" que describe el nombre del comando (o función es una página de la sección 2 o 3) y una descripción en una sola línea del mismo.

La sección siguiente (*synopsis*) describe una guía rápida de uso (que es útil sólo si sabes utilizar ya el comando, o función, y quieres recordar algún argumento). Y a continuación puedes encontrar la descripción detallada del comando o función.

Si sigues avanzando, verás cerca del final otra sección denominada "*see also*" (ver también), que menciona otras páginas de manual relacionadas con la que estás leyendo. Esta sección es muy útil para descubrir otros comandos o llamadas relacionados con lo que estás haciendo.

La página de manual que acabamos de ver mencionará dos comandos que te resultarán muy útiles: *whatis* y *apropos*. El primero puedes utilizarlo para averiguar qué es un comando o función. Por ejemplo:

```
unix$ whatis man
man(1) - display manual pages
man.conf(5) - configuration file for man 1
man(7) - legacy formatting language for manual pages
```

Y puedes ver qué hace el comando *man*. Como hay varias páginas para *man*, *whatis* ha enumerado las que conoce.

El comando *apropos* lo puedes utilizar para buscar comandos y funciones casi del mismo modo que utilizas un buscador en internet. Por ejemplo, para ver cómo compilar nuestro programa en C...

```
unix$ apropos compiler
c++(1) - GNU project C and C++ compiler
cc(1) - GNU project C and C++ compiler
gencat(1) - NLS catalog compiler
rpcgen(1) - RPC protocol compiler
zic(8) - time zone compiler
```

La segunda página tiene buen aspecto, y podemos ahora ejecutar...

```
unix$ max cc
...
```

para ver la página de manual del compilador.

Cada vez que veas un comando en este curso, puedes utilizar el manual para ver cómo se utiliza. Lo mismo sucede con las llamadas que hamos en C. Si algunos trozos de las páginas de manual resultan difíciles de entender no hay que preocuparse. Puedes ignorar esas partes y buscar la información que te interesa. Una vez completes este curso no deberías tener problema en entender el manual.

Miremos ahora la página de un comando que ya hemos utilizado, *uname(1)*:

```
unix$ man uname
NAME
     uname - print operating system name

SYNOPSIS
     uname [-amnprsv]

DESCRIPTION
     The uname utility writes symbols representing one or more system
     characteristics to the standard output.

     The options are as follows:

     -a      Behave as though all of the options -mnrsv were specified.
...
```

El epígrafe *synopsis* muestra cómo utilizar el comando de forma rápida. Los argumentos (y opciones) que aparecen entre corchetes son opcionales. Así pues, podemos ejectuar

```
uname
```

o bien

```
uname -a
```

o quizá

```
uname -m
```

etc. El epígrafe *description* muestra habitualmente el significado de cada una de las opciones y podemos utilizarlo para ver qué uso tenía cada opción o para buscar una opción que produzca el efecto que queremos. Cuando buscamos un comando para hacer algo, resulta útil buscar un comando que haga algo similar o que tenga algo que ver y mirar si hay alguna opción que quizá consiga lo que andamos buscando.

Si miramos ahora la página de *who(1)* podemos aprender algo más:

```
unix$ man who
NAME
     who - display who is logged in

SYNOPSIS
     who [-HmqTu] [file]
     who am i
...
```

Esta vez la synopsis muestra varias líneas. Normalmente eso quiere decir que podemos utilizar el comando de cualquiera de esas formas, pero indica que no podemos combinar ambas en un sólo uso.

Por ejemplo, la segunda línea es precisamente

```
who am i
```

lo que indica que en este caso no se espera que podamos utilizar opciones. No obstante, la primera línea indica que podemos utilizar `who` con la opción `-H`, con la opción `-m`, etc.

Las opciones pueden indicarse por separado como en

```
ls -l -a
```

o pueden indicarse en un sólo argumento, como en

```
ls -al
```

o

```
ls -la
```

El efecto suele ser el mismo.

7. Utilizando ficheros

Antes de continuar y utilizar UNIX para escribir nuestros programas, resulta útil aprender algunos comandos y ver cómo utilizar el shell. Como hemos visto, podemos escribir líneas terminadas en un *enter* (llamado también fin de línea) para escribir comandos. Siempre que las escribamos en un terminal que ejecute un intérprete de comandos. Por ejemplo, para ver la fecha:

```
unix$ date
Wed May  4 17:32:32 CEST 2016
```

Como ya dijimos, "`unix$`" es el prompt del shell y nosotros hemos escrito "`date`" y pulsado *intro* a continuación. El shell ha ejecutado *date*, y dicho comando ha escrito la fecha y hora.

Para listar ficheros puedes utilizar el comando *ls*.

```
unix$ ls
bin    lib    tmp
```

O listar no sólo el nombre, sino también el tipo de fichero, permisos, dueño, fecha en que se modificaron y el nombre. A esto se le llama un listado largo:

```
unix$ ls -l
total 0
drwxr-xr-x  2 nemo  wheel  136 May  3 18:21 bin
drwxr-xr-x  2 nemo  wheel   68 May  3 16:31 lib
drwxr-xr-x  2 nemo  wheel  170 May  3 17:13 tmp
```

En este caso, hemos utilizado "`-l`" como argumento de *ls*. Este argumento produce un listado largo. En realidad, "`-l`" es una opción para *ls*. Si miras la página de manual *ls(1)* verás que la sinopsis muestra argumentos que comienzan por un "`-`" y muchas veces son un sólo carácter. Cada uno de estos caracteres se pueden utilizar como un interruptor o *flag* para modificar el comportamiento del comando. En nuestro caso, hemos activado el flag "`l`" utilizando la opción "`-l`". Sencillamente, los primeros argumentos que utilizamos al ejecutar un comando pueden ser opciones (que empiezan por un "`-`" y siguen con los caracteres de las opciones). Por ejemplo, la opción "`k`" de *ls* utiliza tamaños en Kbytes y la opción "`s`" muestra el tamaño para cada fichero. Sabiendo esto...

```
unix$ ls -ks /bin/ls
16 /bin/ls
```

vemos que el fichero `/bin/ls` contiene 16 Kbytes. Como verás, puedes decir a *ls* qué ficheros quieres listar escribiendo su nombre como argumentos.

Podríamos haber ejecutado:

```
unix$ ls -k -s /bin/ls
16 /bin/ls
unix$ ls -s -k /bin/ls
16 /bin/ls
unix$ ls -sk /bin/ls
16 /bin/ls
```

Pero fíjate en esto...

```
unix$ ls /bin/ls -sk
ls: -sk: No such file or directory
/bin/ls
```

Esta vez, hemos escrito el argumento "`/bin/ls`" antes de "`-sk`". Dado que dicho argumento no empieza por "`-`", *ls* entiende que se refiere al fichero o directorio que queremos listar. ¡Y entiende que no hay mas opciones! Cuando intenta listar el fichero "`-sk`", ve que dicho fichero no existe y escribe un mensaje de error para informarnos de ello. Las opciones debe estar antes del resto de argumentos.

Hay dos opciones más que resultan útiles con `ls`. La primera es `-a`, que hace que `ls` muestre también los ficheros cuyo nombre comienza por un "`.`", cosa que normalmente no hace `ls`. Estos ficheros suelen utilizarse para guardar configuración para diversos programas y normalmente no se desea listarlos, pero pueden estar en cualquier directorio y no los veremos salvo que utilicemos el flag `-a`.

La segunda opción es `-d`, que hace que `ls` liste la información de un directorio en sí mismo y no la de los ficheros que contiene, cuando pidamos a `ls` que liste un directorio.

Si quieres escribir más de un comando en una línea, puedes separarlos por un carácter "`;`", como en:

```
unix$ date ; ls
Wed May  4 17:36:55 CEST 2016
bin    lib    tmp
```

En otras ocasiones queremos escribir líneas muy largas y podemos utilizar un "\" justo antes de pulsar *intro* para *escapar* el fin de línea. El carácter "\" (*backslash*) se utiliza en el shell para quitarle el significado especial a caracters tales como *intro*, que habitualmente tienen significado para el shell (en este caso, ejecutar el comando terminado por el fin de línea). Por ejemplo:

```
unix$ date ;\
> date ;\
> date
Wed May  4 17:40:19 CEST 2016
Wed May  4 17:40:19 CEST 2016
Wed May  4 17:40:19 CEST 2016
```

Hemos pulsando *intro* tras cada "\" y tras el último "`date`". Como verás, el shell ha leído las tres líneas antes de ejecutarlas. Tras cada línea el prompt ha cambiado a "`>`" para indicarnos que el shell está leyendo una nueva línea como continuación del comando. Habría dado igual si ejecutamos:

```
unix$ date ; date ; date
...
```

Otro comando realmente útil, a pesar de lo poco que hace, es *echo*. Este comando hace eco. Se limita a escribir sus argumentos separados por espacios en blanco y terminados por un salto de línea. Por ejemplo:

```
unix echo$ uno y otro
uno y otro
```

¿Qué sucede si hacemos eco de un carácter especial como el "`;`"? Probemos...

```
unix$ echo uno ; otro
uno
-sh: otro: command not found
```

El shell ha visto el "`;`" y ha ejecutado dos comandos. Uno era *echo* y el otro comando era "`otro`", que no existe. Como no existe, el shell nos ha informado del error. Pero podemos ejecutar esto otro...

```
unix$ echo uno \; otro
uno ; otro
```

Como puedes ver, el *backslash* le quita el significado especial al "`;`". Así pues, el shell ejecuta un único comando (*echo*) con tres argumentos. Y *echo* se limita a hacer eco de ellos.

Puedes utilizar *echo* para ver qué argumentos reciben los comandos, como has comprobado en este ejemplo. Existen otras formas de pedir al shell que tome texto literalmente como una sóla palabra, sin que tenga significado especial ninguno de sus caracteres. La más simple es encerrar en comillas simples dicho texto. Por ejemplo:

```
unix$ echo 'uno ; otro'
uno ; otro
```

hace que el shell ejecute un sólo comando, *echo*. Esta vez, el comando recibe un único argumento (y no tres como antes). El argumento contiene el texto que hay entre comillas simples.

La opción "`-n`" de *echo* hace que *echo* no escriba el salto de línea tras escribir los argumentos. Por ejemplo, ejecutando

```
unix$ echo -n hola
holaunix$
```

la salida de *echo* ("`hola`") aparece justo antes del prompt del shell para el siguiente comando. Simplemente nadie ha escrito el salto de línea y el shell se ha limitado a escribir el prompt. Otro ejemplo:

```
unix$ echo a ; echo b
a
b
unix$ echo -n a ; echo  b
ab
unix$
```

Fíjate como el penúltimo *echo* ha escrito su argumento pero no ha saltado a la siguiente línea.

Podemos crear un fichero utilizando el comando *touch*. Por ejemplo:

```
unix$ ls
bin    lib    tmp
unix$ touch fich
unix$ ls
bin    fich    lib    tmp
```

Hemos utilizado el *argumento* "`fich`" para el comando *touch*. Como no existe dicho fichero, el comando lo crea vacío. Una vez creado, el comando *ls* lo ha listado.

Otra forma útil de crear fichero pequeños es utilizar *echo* y pedirle al shell que lo engañe para que escriba su salida en un fichero. Por ejemplo:

```
unix$ echo hola >fich
```

Ejecuta *echo* pero envía su salida al fichero `fich`. El comando no sabe dónde está escribiendo. El "`>`" le indica al shell que queremos que le pida a UNIX que la salida de *echo* se escriba en el fichero indicado.

Podemos ver el contenido del fichero utilizando el comando *cat*, que escribe el contenido de los ficheros que le indicamos como argumento:

```
unix$ cat fich
hola
```

Es posible copiar un fichero en otro utilizando *cp*:

```
unix$ cp fich otro
cp fich otro
unix$ ls
bin    fich    lib    otro    tmp
```

Y ahora podemos borrar el fichero antiguo utilizando *rm*:

```
unix$ rm fich
unix$ ls
bin    lib    otro    tmp
```

La mayoría de los comandos capaces de aceptar un nombre de fichero como argumento son capaces de aceptar varios. Por ejemplo:

```
unix$ touch a b c
unix$ ls
a    b    bin    c    lib    otro    tmp
unix$ rm a b c
unix$ ls
bin    lib    otro    tmp
```

Además, hay comandos como *ls* que utilizan un valor por omisión cuando no reciben un nombre de fichero como argumento. Por ejemplo, *ls* lista el directorio en el que estás si no indicas qué ficheros quieres listar, como has visto antes.

Es importante escribir los nombres de fichero exactamente. Casi todos los sistemas UNIX son sensibles a la capitalización (son *case sensitive*). Por ejemplo:

```
unix$ touch Uno
unix$ rm uno
rm: uno: No such file or directory
```

El comando *rm* no encuentra el fichero `uno`. Pero...

```
unix$ rm Uno
```

funciona perfectamente.

Otro comando útil es *mv*. Se utiliza para mover (y renombrar) ficheros. Por ejemplo, podemos utilizar

```
unix$ mv fich otro
```

en lugar de

```
unix$ cp fich otro ; rm fich
```

Antes de seguir utilizando *cp*, *mv* y otros comandos relacionados con fichero necesitamos ver qué son los directorios y como se utilizan.

8. Directorios

Igual que la mayoría de sistemas, UNIX utiliza directorios para agrupar ficheros. Usuarios de Windows suelen llamar "carpeta" a los directorios. Un directorio es tan sólo un fichero que agrupa varios ficheros. Aparentemente, un directorio contiene otros ficheros y directorios, pero esto es otra ilusión implementada por el sistema operativo. Todos los ficheros y directorios están guardados en el disco (en la mayoría de los casos), por separado. No obstante, podemos listar el contenido de un directorio y veremos una lista de ficheros (y directorios). Dos ficheros en dos directorios distintos siempre serán ficheros distintos aunque tengan el mismo nombre.

En pocas palabras, los ficheros están agrupados en un árbol cuya raíz es un directorio (el directorio raíz, llamado "`/`" o *slash* en UNIX). Dentro del raíz tenemos ficheros y directorios, y así sucesivamente. La figura 3 muestra parte del árbol de ficheros que puedes ver en un sistema UNIX.
Tendrás muchos mas ficheros en cualquier sistema que utilices. El directorio raíz (`/`) contiene directorios llamados `bin`, `home`, `tmp`, `usr`, y otros que no mostramos. El directorio `bin` suele contener el binario de comandos del sistema. El directorio `home` suele contener un directorio por cada usuario, para que dicho usuario pueda dejar sus ficheros dentro del mismo. El directorio `tmp` suele utilizarse para crear ficheros temporales, que queremos utilizar sólo durante un rato y borrar después. En nuestro ejemplo, el usuario *nemo* ha creado directorios llamados `bin`, `lib`, y `tmp` dentro del directorio `nemo`.

Cada fichero o directorio tiene un nombre, escrito con texto en la figura. Pero para localizar un fichero (o directorio) hay que decirle a UNIX cuál es el camino en el árbol de ficheros para llegar hasta el fichero en

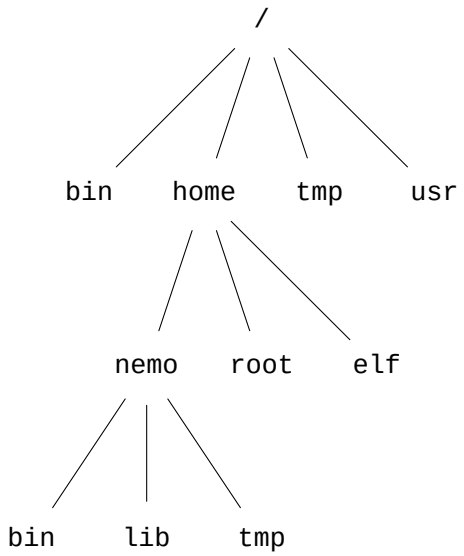

Figura 3: *Ejemplo de árbol de ficheros en UNIX.*

cuestión. Por ejemplo, el camino para el directorio del usuario *nemo* sería "`/home/nemo`". Hemos escrito el cambio, o *path*, empezando por el directorio raíz, "`/`", y nombrando los directorios según bajamos por el árbol, separados unos de otros por un "`/`". La última componente del path es el nombre del fichero.

Puedes ver que hay dos directorios llamados `tmp`. Uno es `/tmp` (el que está directamente dentro del raíz) y otro es `/home/nemo/tmp`. Esto es: Empezamos en el raíz, y seguimos por `home`, luego `nemo` y finalmente `tmp`. Ambos paths se denominan absolutos dado que comienzan por el raíz e identifican un fichero (o directorio) sin duda alguna.

Sería incómodo tener que escribir paths absolutos cada vez que queremos nombrar un fichero. Debido a esto, cada programa que ejecuta en UNIX está asociado a un directorio. Dicho directorio se denomina *directorio actual* o *directorio de trabajo* del programa en ejecución que consideremos.

Cuando entramos al sistema y se ejecuta un shell para que podamos ejecutar comandos, el shell utiliza como directorio de trabajo un directorio que el administrador creó al crear nuestro usuario en UNIX (al darnos de alta como usuario en el sistema o crear nuestra cuenta). Para el usuario *nemo*, ese directorio podría ser `/home/nemo`. Decimos que dicho directorio es el directorio *casa* (o *home*) de dicho usuario.

Pues bien, utilizando el árbol del la figura, el usuario *nemo* podría cambiar el directorio de trabajo del shell a otro directorio utilizando el comando que sigue:

```
unix$ cd /tmp
```

En este caso, el directorio ha pasado a ser `/tmp`.

¡Y ahora podemos crear `/tmp/a` como sigue!

```
unix$ touch a
```

No ha sido preciso ejecutar

```
unix$ touch /tmp/a
```

Dado que *touch* le ha pedido a UNIX que cree el fichero "`a`", y que el nombre de fichero (el path) no comienza por "`/`", UNIX entiende que el path hay que recorrerlo a partir del directorio actual o directorio

de trabajo. El comando *touch* ha "heredado" el directorio actual del shell, como se verá más adelante. Dado que el directorio era "`/tmp`", el path resultante es "`/tmp/a`".

Si piensas en los comandos que hemos ejecutado anteriormente, verás ahora como los paths que hemos utilizado se han resuelto a partir del directorio de trabajo. Estos paths se denominan *relativos* (no empiezan desde el raíz).

¿Y en qué directorio estamos? Es fácil verlo, utilizando el comando *pwd* que escribe su directorio de trabajo.

```
unix$ pwd
/tmp
```

Y podemos cambiar el directorio de trabajo del shell utilizando el comando *cd*:

```
unix$ cd /
unix$ pwd
/
```

Hay dos nombres de fichero (de directorio, en realidad) que existen en cada directorio del árbol de ficheros. Uso se llama "`.`" y el otro "`..`". Corresponden al directorio en que están y al directorio que está arriba en el árbol, respectivamente. Así pues,

```
unix$ cd .
```

no hace nada. Si el directorio actual es `/tmp`, "`.`" significa `/tmp/.`, lo cual significa `/tmp`. Así que el comando *cd* deja el directorio actual en el mismo sitio.

Si ahora ejecutamos...

```
unix$ pwd
/tmp
unix$ cd ..
unix$ pwd
/
```

vemos que "`..`" en `/tmp` corresponde al directorio raíz. Si seguimos insistiendo...

```
unix$ cd ..
unix$ pwd
/
```

vemos que no es posible subir más arriba en el árbol. No hay nada fuera del raíz.

Es importante entener que cada programa que ejecuta tiene su propio directorio de trabajo. Por ejemplo, podemos ejecutar un shell y cambiar su directorio

```
unix$ pwd
/home/nemo
unix$ sh
unix$ cd /
unix$ pwd
/
```

Ahora, si escribimos el comando *exit*, que indica al shell que termine, volveremos al shell que teníamos al principio, cuyo directorio era `/home/nemo`.

```
unix$ exit
unix$ pwd
unix$ /home/nemo
```

El directorio *home* es tan importante (para cada usuario el suyo) que el comando *cd* nos lleva a dicho directorio (cambia su directorio de trabajo al directorio casa) si no le damos argumentos. Es fácil averiguar cuál es tu directorio *home*:

```
unix$ cd
unix$ pwd
/home/nemo
```

Dispones del comando *mkdir* para crear nuevos directorios. Por ejemplo, es probable que el usuario *nemo* ejecutó un comando como

```
unix$ mkdir bin
```

para crear el directorio `/home/nemo/bin` que viste en el árbol de ficheros de la figura 3.

Puedes borrar directorios utilizando el comando *rmdir*. Por ejemplo,

```
unix$ rmdir /home/nemo/bin
```

borraría el directorio `bin`. Esta vez hemos utilizado un path absoluto.

Los comandos *cp* y *mv* se comportan de un modo diferente cuando el destino es un directorios. En tal caso, copian o mueven los ficheros origen hacia ficheros en dicho directorio. Por ejemplo, si empezamos en un directorio vacío:

```
unix$ mkdir fich
unix$ touch a b
unix$ ls
a    b    fich
unix$ cp a b fich
unix$ ls fich
a    b
```

Esto es, *cp* ha copiado tanto `a` como `b` a ficheros `fich/a` y `fich/b`. ¿Entiendes ahora los caminos relativos?

Igualmente podemos mover uno o varios ficheros a un directorio:

```
unix$ mv a b fich
unix$ ls
fich
unix$ ls fich
a    b
```

Si ahora intentamos borrar el directorio...

```
unix$ rmdir fich
rmdir: fich: Directory not empty
```

el comando *rmdir* no se deja. Si se dejase, ¿Qué pasaría si ejecutas por accidente el siguiente comando?

```
unix$ rmdir /
```

Tendrías diversión a raudales...

Para borrar un directorio es preciso borrar antes los ficheros (y directorios) que contiene. El comando *rm* tiene la opción `-r` (recursivo) que hace justo eso:

```
unix$ rm -r fich
```

¿Comprendes por qué no debes intentar en casa el siguiente par de comandos?

```
unix$ cd
unix$ rm -r .
```

9. ¿Qué contienen los ficheros?

El UNIX, los ficheros contienen bytes. Eso es todo. UNIX no sabe qué significan esos bytes ni para qué sirven. ¿Recuerdas el fichero fuente en C que escribimos? Podemos ver el contenido del fichero utilizando el comando *cat*:

```
unix$ cat hola.c
#include <stdio.h>

int
main(int argc, char *argv[])
{
    puts("hola\n");
}
```

En realidad *puts* escribe un fin de línea por su cuenta tras escribir el string que le pasamos, por lo que este programa deja una línea en blanco tras escribir "`hola`". Pero también podemos utilizar el comando *od* (octal dump) para que nos escriba el valor de los bytes del fichero:

```
od -c hola.c
0000000    #   i   n   c   l   u   d   e       <   s   t   d   i   o   .
0000020    h   >  \n  \n   i   n   t  \n   m   a   i   n   (   i   n   t
0000040        a   r   g   c   ,       c   h   a   r       *   a   r   g
0000060    v   [   ]   )  \n   {  \n  \t   p   u   t   s   (   "   h   o
0000100    l   a   \   n   "   )   ;  \n   }  \n  \n
0000113
```

En cada línea, *od* escribe unos cuantos bytes indicando al principio de la línea el número de byte (en octal). Cada línea contiene 16 bytes y por tanto la segunda línea comienza en la posición 20 (`2*8`).

La opción `-c` de *od* hace que se escriban los caracteres correspondientes al valor de cada byte. Pero podemos pedirle a *od* que escriba además el valor de cada byte en hexadecimal:

```
unix$ od -c -t x1 hola.c
0000000    #   i   n   c   l   u   d   e       <   s   t   d   i   o   .
          23  69  6e  63  6c  75  64  65  20  3c  73  74  64  69  6f  2e
0000020    h   >  \n  \n   i   n   t  \n   m   a   i   n   (   i   n   t
          68  3e  0a  0a  69  6e  74  0a  6d  61  69  6e  28  69  6e  74
0000040        a   r   g   c   ,       c   h   a   r       *   a   r   g
          20  61  72  67  63  2c  20  63  68  61  72  20  2a  61  72  67
0000060    v   [   ]   )  \n   {  \n  \t   p   u   t   s   (   "   h   o
          76  5b  5d  29  0a  7b  0a  09  70  75  74  73  28  22  68  6f
0000100    l   a   \   n   "   )   ;  \n   }  \n  \n
          6c  61  5c  6e  22  29  3b  0a  7d  0a  0a
0000113
```

Si te fijas en el fichero mostrado con *cat* y lo comparas con la salida de *od* podrás ver qué contiene en realidad el fichero con tu código fuente. Por ejemplo, tras el "`>`" de la primera línea, hay un byte cuyo valor es

10 (o `0a` escrito en hexadecimal). Dicho byte se muestra como "`\n`" si hemos pedido a *od* que escriba cada byte como un carácter. En el caso de "`\n`", se trata del carácter que corresponde al fin de línea. Es ese el carácter que hace que la segunda línea se muestre en otra línea cuando *cat* escribe el fichero en el terminal (en la pantalla o en la ventana). Si miras ahora el fuente, verás que hay una línea en blanco tras el *include*. En la salida de *od* vemos que simplemente no hay nada entre el `\n` que termina la primera línea y el `\n` que termina la segunda.

Otro detalle importante es que no existe ningún carácter que marque el fin del fichero. No existe *eof*. Igual que un libro, un fichero se termina cuando no tiene más datos que puedas leer.

El fuente estaba tabulado, utilizando el carácter tabulador, ("`\t`", cuyo valor es 9). Al mostrar ese carácter, el terminal avanza hasta que la columna en que escribe es un múltiplo de 8 (puede cambiarse en ancho del tabulador). Por eso se llama *tabulador*, por que sirve para tabular o escribir tablas o texto con forma de tabla. Hoy día se utiliza para sangrar el fuente más que para otra cosa.

Tanto el fin de línea, como el tabulador y otros caracteres especiales, son especiales sólo por que los programas que los utilizan les dan un significado especial. Pero no tienen nada de especial. Siguen siendo un byte con un valor dado. Eso si, si un editor o un terminal muestra un "`\n`", entiende que ha de seguir mostrando el texto en la siguiente línea. Y si muestra un "`\t`", el programa entenderá que hay que avanzar para tabular. A estos caracteres se los denomina caracteres de control.

Otro carácter de control es "`\b`", o *backspace* (borrar), que hace que se borre el carácter anterior. De nuevo, es el programa que utiliza el carácter el que hace que sea especial. Por ejemplo, al escribir una línea de comandos, "`\b`" borra el carácter anterior. ¡Pero no puedes borrar un *intro* si lo has escrito! Cuando escribes el fin de línea, UNIX le da el texto al shell, con lo que UNIX (el kernel) no puede cancelar el último carácter que escribiste antes de borrar. Eso si, si pulsas un carácter y luego *backspace*, el kernel tira ambos caracteres a la basura (y actualiza el texto que ves en el terminal). La tecla de borrar borra también en un editor de texto sólo debido a que el programa del editor interpreta que quieres borrar, y actualiza el texto que hay escrito.

En este texto, y en la salida de comandos como *od*, se utiliza la sintaxis del lenguaje C para escribir caracteres de control: Un *backslash* y una letra.

Hay algo más que debes saber sobre ficheros que contienen texto. Hace tiempo se codificaba cada carácter con un único byte, empleando la tabla ASCII (7 bits por carácter). El comando *ascii* existe todavía e imprime la tabla:

```
unix$ ascii
|00 nul|01 soh|02 stx|03 etx|04 eot|05 enq|06 ack|07 bel|
|08 bs |09 ht |0a nl |0b vt |0c np |0d cr |0e so |0f si |
|10 dle|11 dc1|12 dc2|13 dc3|14 dc4|15 nak|16 syn|17 etb|
|18 can|19 em |1a sub|1b esc|1c fs |1d gs |1e rs |1f us |
|20 sp |21  ! |22  " |23  # |24  $ |25  % |26  & |27  ’ |
|28  ( |29  ) |2a  * |2b  + |2c  , |2d  - |2e  . |2f  / |
|30  0 |31  1 |32  2 |33  3 |34  4 |35  5 |36  6 |37  7 |
|38  8 |39  9 |3a  : |3b  ; |3c  < |3d  = |3e  > |3f  ? |
|40  @ |41  A |42  B |43  C |44  D |45  E |46  F |47  G |
|48  H |49  I |4a  J |4b  K |4c  L |4d  M |4e  N |4f  O |
|50  P |51  Q |52  R |53  S |54  T |55  U |56  V |57  W |
|58  X |59  Y |5a  Z |5b  [ |5c  \ |5d  ] |5e  ^ |5f  _ |
|60  ‘ |61  a |62  b |63  c |64  d |65  e |66  f |67  g |
|68  h |69  i |6a  j |6b  k |6c  l |6d  m |6e  n |6f  o |
|70  p |71  q |72  r |73  s |74  t |75  u |76  v |77  w |
|78  x |79  y |7a  z |7b  { |7c  | |7d  } |7e  ~ |7f del|
```

Compara el fuente y la salida de *od* con los valores de la tabla.

Pero vamos a crear un fichero con el carácter " ".

```
unix$ echo   >fich
```

Y ahora vamos a ver qué contiene el fichero:

```
unix$ cat fich

unix$ od -c fich
0000000  316 261  \n
0000003
```

¡Contiene 3 bytes! En muchos casos, caracteres con tilde como "ñ" o "ó", o letras como " ", y otras muchas no pueden escribirse en ASCII. Eso hizo que se utilizasen otros formatos para codificar texto. Hoy día suele utilizarse Unicode como formato para guardar caracteres. Cada carácter, llamado *runa* o *code-point* en Unicode, corresponde a un valor que necesita en general varios bytes. Para guardar ese valor se utiliza un formato de codificación que hace que para las runas que existían en ASCII se utilice el mismo valor que en ASCII (por compatibilidad hacia atrás con programas que ya existían), y para las otras se utilicen más bytes. De ese modo, "`\n`" sigue siendo un byte con valor 10 (o `0a` en hexadecimal), pero " " no. Si utilizamos como formato de codificación UTF-8, corresponde con los bytes `ce b1`:

```
unix$ od -t x1 fich
0000000   ce  b1  0a
0000003
```

En cualquier caso, siguen siendo bytes. Tan sólo recuerda que cada sistema utiliza una tabla de codificación de caracteres que asigna a cada carácter un valor entero. Y para sistemas como Unicode, tienes además un formato de codificación que hace que sea posible escribir cada valor como uno o más bytes. Se hace así para para que el texto que puede escribirse en ASCII siga codificado como en ASCII, para permitir que los programas antiguos sigan funcionando igual.

Para UNIX, los ficheros contienen bytes. Pero para los programas que manipulan ficheros con texto y para los humanos los ficheros de texto contienen bytes que corresponden a UTF-8 o algún otro formato de texto.

Los directorios contienen también bytes. Por cada fichero, contienen el nombre del fichero en cuestión y algo de información extra que permite localizar ese fichero en el disco. Pero para evitar que los usuarios y los programas rompan los datos que se guardan en los directorios, y causen problemas al kernel cuando los usa, UNIX no permite leer o escribir directamente los bytes que se guardan en un directorio. Hace tiempo lo permitía, pero los bugs en programas de usuario causaban muchos problemas y decidieron que era mejor utilizar otras llamadas al sistema para directorios, en lugar de las que se utilizan para leer y escribir ficheros. Veremos esto cuando hablemos de ficheros más adelante.

10. Usuarios y permisos

Cuando vimos cómo entrar al sistema, y al utilizar comandos para listar ficheros y directorios, hemos visto que UNIX tiene idea de que hay usuarios en el sistema. La idea de "*usuario*" es de nuevo una abstracción del sistema. Concretamente, cada programa que ejecuta lo hace a nombre de un usuario. Dicho "nombre" es un número para UNIX. El comando *id* muestra el identificador de usuario, o *user id*, o *uid*.

```
unix$ id
uid=501(nemo) gid=20(staff) groups=20(staff)
```

Para crear un usuario en UNIX normalmente editamos varios ficheros en `/etc` para informar al sistema del nuevo usuario. Esto suele hacerse utilizando comandos (descritos en la sección 8 del manual) para evitar

cometer errores. Concretamente, hay que indicar al menos el nombre de usuario (por ejemplo, "`nemo`"), el número que utiliza UNIX como nombre para el usuario (por ejemplo, 501)., y qué directorio utiliza ese usuario como casa.

Además, habrá que crear el directorio casa para el usuario y darle permisos para que pueda utilizarlo. También se suele indicar qué shell prefiere utilizar dicho usuario, por ejemplo, `/bin/sh`, y otra información administrativa incluyendo el nombre real del usuario.

En UNIX, los usuarios pertenecen a uno o más grupos de usuarios. Una vez más, para UNIX, un grupo de usuarios es un número. En nuestro caso, *nemo* pertenece al grupo *staff* (y UNIX lo conoce como 20). El número que identifica al grupo de usuarios se denomina *group id* o *gid*. Al crear un usuario también hay que decir a qué grupos pertenece.

Aunque la forma de crear un usuario varía mucho de un UNIX a otro, este comando se utiliza en OpenBSD:

```
unix# adduser
Enter username []: loser
Enter full name []: Mr Loser
Enter shell bash csh ksh nologin rc sh [ksh]: ksh
Uid [1005]:
Login group loser [loser]: staff
Login group is ‘‘staff’’. Invite loser into other groups: guest no
[no]:
Enter password []:
Enter password again []:
```

Su efecto es editar el fichero de cuentas `/etc/passwd` (y otros ficheros) para guardar la información del usuario, de tal forma que *login* y otros programas puedan encontrarla. Por ejemplo, esta línea aparece en el fichero de cuentas tras crear el usuario:

```
loser:*:1005:50:Mr Loser:/home/loser:/bin/ksh
```

Además, el comando ha creado el directorio `/home/loser` y ha dado permisos al nuevo usuario para utilizarlo.

De hecho, los ficheros y directorios pertenecen a un usuario (UNIX guarda el *uid* del dueño) y a un grupo de usuarios (UNIX guarda el *gid*). El comando *adduser* ha hecho que `/home/loser` pertenezca al usuario `loser`, y al grupo `staff` en nuestro ejemplo.

Los programas en ejecución están ejecutando a nombre de un usuario (con un *uid* dado). Cuando un programa crea ficheros, estos ficheros se crean a nombre del usuario que ejecuta el programa.

Para realizar cualquier operación o llamada al sistema el usuario debe tener permiso para ello. Por ejemplo, normalmente no podrás ejecutar *adduser* para crear usuarios. Las llamadas al sistema que utiliza y los ficheros que edita no es probable que estén accesibles con tu usuario. En UNIX, el usuario número 0 (con *uid* 0) es especial. El kernel incluye cientos de comprobaciones que hacen que dicho usuario pueda efectuar la llamada que haga y que, en otro caso, se compruebe si el usuario tiene permiso para hacerla. Por eso a ese usuario se le llama *superusuario*. Su nombre suele ser *root*, pero UNIX lo conoce como *uid* 0. Recuerda que el nombre es tan sólo texto que utilizan programas como *ls* para mostrar el nombre de usuario. El kernel utiliza *uids*. El prompt del shell suele cambiar si el usuario es *root*, para avisarte de ello. En nuestro ejemplo el prompt era "`unix#`" y no "`unix$`".

Podemos cambiar de un usuario a otro utilizando el comando *su*, (*switch user*) que ejecuta en shell a nombre de otro usuario. Por ejemplo,

```
unix$ id
id=501(nemo) gid=20(staff) groups=20(staff)
unix$ su
Password: *******
unix# id
uid=0(root) gid=0(wheel) groups=0(wheel), 20(staff)
unix# adduser
...
unix# exit
unix$ id
id=501(nemo) gid=20(staff) groups=20(staff)
```

Cada fichero en unix incluye información respecto a qué cosas puede hacer qué usuario con el mismo. A esto se le llama *lista de control de acceso*, (o ACL, por *access control list*). Normalmente, UNIX guarda 9 bits al menos correspondientes a 9 permisos para cada fichero: 3 permisos para el dueño del fichero, otros 3 para cualquier usuario que no sea el dueño, pero que pertenezca al grupo al que pertenece el fichero y otros 3 permisos para cualquier otro usuario. Estos bits se guardan en la estructura de datos que utiliza el sistema operativo para cada fichero. Pues bien, los permisos son *lectura*, *escritura* y *ejecución*. Si miramos el listado largo de un fichero podemos verlos:

```
unix$ ls -l fich
-rwxr-xr-- 1 nemo staff 1024 May 30 16:31 fich
```

Concretamente, "`rwxrwxr-x`" son los 9 bits para permisos a los que nos hemos referido. Hay más bits y más permisos, pero los veremos más adelante. En el fichero que hemos listado, `fich`, los tres primeros permisos (esto es, "`rwx`") se aplican al dueño del fichero (esto es, a *nemo*, tal y como indica *ls*). Dicho de otro modo, cualquier programa que ejecute a nombre de *nemo* e intente hacer llamadas al sistema para utilizar el fichero, estará sujeto a los permisos `rwx`. Para cualquier usuario que, no siendo el dueño, pertenezca al grupo *staff*, se aplicarán los siguientes tres permisos: `r-x`. Y para cualquier otro usuario, se aplicarán los últimos tres permisos: `r--`.

En cualquiera de los casos, la "`r`" indica permiso de lectura, la "`w`" indica permiso de escritura y la "`x`" indica permiso de ejecución. En un fichero, leer el contenido de fichero requiere permiso de lectura. Los comandos como *cat* y *cp* leen el contenido de ficheros y requieren dicho permiso. Escribir el contenido del fichero require permiso de escritura. Por ejemplo,

```
unix$ echo hola >fich
```

require permiso de escritura en `fich`. El permiso de ejecución se utiliza para permitir que un fichero se ejecute (por ejemplo, en respuesta a un comando que escribimos en el shell).

En el caso de los directorios, la "`r`" indica permiso para listar el contenido del directorio, o leer el directorio. La "`w`" indica permiso para modificar el directorio, ya sea creando ficheros dentro o borrándolos o cambiando su nombre. Y la "`x`" indica permiso para entrar en el directorio (cambiando de directorio dentro de él, por ejemplo).

Para modificar los permisos se utiliza el comando *chmod*, (*change mode*). Si primer argumento se utiliza para dar o quitar permisos y el resto de argumentos corresponden a los ficheros a los que hay que ajustarle los permisos. Por ejemplo,

```
unix$ chmod +x fich
```

hace que `fich` sea ejecutable. Para UNIX, cualquier fichero con permiso de ejecución es un ejecutable. Otra cosa será si cuando se intente ejecutar su contenido tiene sentido o no.

Para evitar escribir un fichero por accidente, podemos quitar su permiso de escritura:

```
unix$ chmod -w fich
```

Cuando utilizamos "+", damos permiso. Cuando utilizamos "-", lo quitamos. Detrás podemos escribir uno o más permisos:

```
unix$ chmod +rx fich
```

En ocasiones, queremos dar permisos sólo para el usuario que es propietario del fichero:

```
unix$ chmod u+w fich
```

La "u" quiere decir *user*. Igualmente, podemos quitar un permiso sólo para el grupo de usuarios al que pertenece el fichero:

```
unix$ chmod g-w fich
```

La "g" quiere decir *group*. Del mismo modo podemos usar inicialmente una "o" (por *others*) para cambiar los permisos para el resto de usuarios.

En otros casos, resulta útil utilizar como argumento para los permisos el número que guarda UNIX para los mismos. Igual sucede al programar en C, donde hemos de utilizar una llamada al sistema que acepta un entero para indicar los permisos que deseamos. Es fácil si pensamos que estos nueve bits son tres grupos de tres bits. Cada grupo de permisos puede ir de "000" a "111". Escribiendo en base 8 podemos utilizar un sólo dígito para cada grupo de bits, y es justo lo que se hace. La figura 4 muestra el esquema. El comando

```
unix$ chmod 755 fich
```

dejaría los permisos de `fich` como "rwxr-xr-x", dado que 7 es 111 en binario, con los tres bits a 1. Además, el segundo y tercer dígito en octal valen 5, por lo que los permisos para el grupo serían "r-x" (4 mas 1, o 101) y lo mismo los permisos para el resto de usuarios.

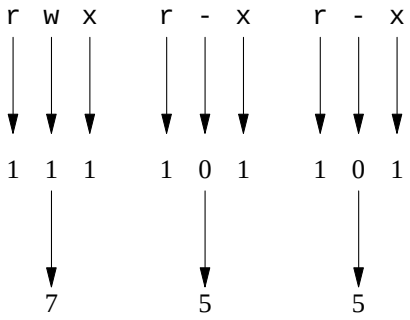

Figura 4: *Los permisos en UNIX se codifican como un número en octal en muchas ocasiones.*

11. Compilado y enlazado

Podemos ver con más detalle qué sucede cuando compilamos, enlazamos y ejecutamos un programa que hace llamadas al sistema.

Volvamos a considerar el programa para escribir un saludo, que reproducimos aquí por comodidad.

[hola.c]:

```
int
main(int argc, char *argv[])
{
    puts("hola");
}
```

Cuando ejecutamos el comilador y generamos un fichero objeto para este fichero fuente, obtenemos el fichero `hola.o`, como vimos antes.

```
unix$ cc hola.o
unix$ ls -l
total 64
-rw-r--r--  1 elf  wheel    75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel  1288 May  4 16:03 hola.o
```

El fichero objeto contiene parte del binario para el programa ejecutable. La parte que corresponde al fuente que hemos compilado. También incluye información respecto a qué símbolos (nombres para código y datos) incluye el fichero objeto y qué símbolos necesita de otros ficheros para terminar de construir un ejecutable.

Dado que es muy tedioso intentar entender el binario, lo que podemos hacer es ver el código ensamblador que ha generado el compilador y ha ensamblado para generar el fichero objeto. Esto se puede hacer utilizando el flag `-S` del compilador de C.

```
unix$ cc -S hola.c
unix$ ls -l
total 64
-rw-r--r--  1 elf  wheel    75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel  1288 May  4 16:03 hola.o
-rw-r--r--  1 elf  wheel   840 May  5 09:57 hola.s
```

El fichero `hola.s` contiene el código ensamblador: texto fuente para un programa que traduce dicho código a binario, y que está muy próximo al binario.

`[hola.s]`:

```
        .file   "hola.c"
        .section        .rodata
        .string "hola\n"
        .text
        .globl  main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        leave
        ret
```

Como puedes ver, simplemente se define el valor para un string en la memoria (con el saludo) y hay una serie de instrucciones para main. Todo esto se verá más claro en el siguiente tema. Estas instrucciones se pueden ya codificar como números en binario para que las entienda el procesador (eso es lo que tiene el fichero objeto). Las instrucciones se limitan a situar en la pila los argumentos de *puts* y a llamar a *puts*. El valor para nuestra constante debe ir en una sección de valores para datos y el valor del código de *main* debe ir en una sección de código (texto).

Para que el enlazador sepa enlazar este fichero objeto junto con otros y obtener un fichero ejecutable, el fichero objeto incluye no sólo los valores de los datos inicializados y el código, sino también una tabla que contiene el nombre y posiciones en el fichero para cada símbolo. El comando *nm* lista esa tabla y nos

permite ver lo que contiene un fichero objeto.

```
unix$ nm hola.o
         U _GLOBAL_OFFSET_TABLE_
00000000 F hola.c
00000000 T main
         U puts
```

Como verás, existe un símbolo de tipo "`F`" (fichero fuente) que nombra el fichero del que procede este código. Compara con la directiva "`.file`" que aparece en el listado en ensamblador. Además, aparece un símbolo de código o texto ("`T`") para la función *main*. Por último, hay una anotación respecto a que este código necesita un símbolo llamado "`puts`" para poder construir un ejecutable. Esto es así puesto que *main* llama a *puts*.

El enlazador tomará este fichero objeto y la librería de C, que contiene más ficheros objeto, y construirá un único fichero ejecutable. Para ello irá asignando direcciones a cada símbolo y copiando los símbolos que se necesiten al fichero ejecutable. Las direcciones asignadas serán aquellas que deberán utilizar en la memoria los bytes de cada símbolo.

Por ejemplo, *main* aparece con dirección "`00000000`" en la salida de *nm* para `hola.o`. Esto es normal puesto que se trata de un trozo de un ejecutable. La dirección que utilice en la memoria el código de *main* dependerá de dónde lo decida situar el enlazador. Una vez enlazado, el binario se cargará en la memoria utilizando el sistema operativo, de tal forma que las direcciones utilizadas en la memoria deberán coincidir con las que ha asignado el enlazador.

Podemos construir un ejecutable utilizando el compilador para que llame al enlazador del siguiente modo:

```
unix$ cc -static hola.o
```

Y ahora tenemos un ejecutable llamado `a.out`:

```
unix$ cc hola.o
unix$ ls -l
total 64
-rwxr-xr-x  1 elf  wheel  342427 May  5 10:16 a.out*
-rw-r--r--  1 elf  wheel      75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel    1288 May  4 16:03 hola.o
```

Si listamos los símbolos de `a.out` utilizando *nm* podemos ver una salida como la que sigue.

```
unix$ nm -n a.out
00400270 T __init
00400280 T __start
00400280 T _start
00400480 T atexit
004004f4 T main
00400520 T puts
00400710 T exit
......
0040c930 W write
......
```

Esta vez, hemos utilizado el flag `-n` de *nm* para pedirle que liste los símbolos ordenados por dirección de memoria. Como puedes ver, *main* estará en la dirección `004004f4` cuando esté cargado en la memoria, y *puts* en la dirección `00400520`. Esta llamará a *write*, que estará en la dirección `0040c930`. Dado que es una llamada al sistema, este símbolo está declarado de tipo "`W`" en el UNIX que utilizamos, pero habitualmente podrás verlo como "`T`". (La "`W`" significa *símbolo débil* y es similiar a texto pero permite que un fichero objeto contenga un símbolo del mismo nombre para reemplazar al símbolo débil; Puedes ignorar

esto).

En realidad, normalmente no utilizamos la opción "`-static`" al compilar y enlazar. Vamos a enlazar de nuevo:

```
unix$ cc hola.o
unix$ ls -l a.out
-rwxr-xr-x  1 elf  wheel  8570 May  5 10:26 a.out*
```

Esta vez, el ejecutable contiene unos 8Kbytes y no unos 342Kbytes. Veamos que nos dice *nm*:

```
unix$ nm -n a.out
         U puts
         U exit
00000a10 T __init
00000a80 T __start
00000a80 T _start
00000c80 T atexit
00000cf4 T main
00000d20 T __fini
00201000 D __guard_local
00201008 D __data_start
00201008 D __progname
00201010 D __dso_handle
00201148 a _DYNAMIC
00301290 a _GLOBAL_OFFSET_TABLE_
00401360 B _dl_skipnum
00401370 B _dl_searchnum
...
```

La vez anterior, la salida de *nm* tenía muchas mas líneas que ahora (aunque hemos borrado la mayoría para omitir detalles que no importan por el momento). Ahora, la salida de *nm* tiene sólo cuatro o cinco líneas más que las que mostramos.

Lo que sucede es que el programa está enlazado pero no del todo. Puedes ver que *puts* sigue "U" (*undefined*). Cuando este programa esté cargado en la memoria y llegue al punto en que necesita llamar a *puts*, llamará a código que tiene enlazado y que deberá completar el enlazado. Este código forma parte del denominado *enlazador dinámico*.

El proceso es igual que cuando enlazamos de forma estática (en tiempo de compilación si quieres verlo así) el ejecutable. La diferencia es que ahora hemos enlazado los ficheros objeto junto con código que completa el enlazado ya en tiempo de ejecución.

Además, a diferencia de antes, cuando el programa se enlaza ya en tiempo de ejecución, el enlazador dinámico utiliza el sistema operativo para poder compartir librerías ya cargadas en la memoria (enlazadas dinámicamente con otros programas que están ejecutando). Si un programa utiliza por primera vez un símbolo de una librería dinámica que no se ha cargado aún, el sistema operativo la carga en la memoria y el enlazador dinámico puede completar el enlazado.

El comando *ldd* muestra las librerías que necesitará el ejecutable para completar el enlazado, ya en tiempo de ejecución. Así pues:

```
unix$ cc -static hola.o
unix$ ldd a.out
a.out:
not a dynamic executable
unix$ cc hola.o
unix$ ldd a.out
a.out:
        Start            End              Type Open Ref GrpRef Name
        00001af940c00000 00001af941002000 exe  1    0   0      a.out
        00001afbe5731000 00001afbe5c1d000 rlib 0    1   0      /usr/lib/libc.so.78.1
        00001afbfd900000 00001afbfd900000 rtld 0    1   0      /usr/libexec/ld.so
```

Por ahora nos da igual que quiere decir cada columna, pero puedes ver que el ejecutable utilizará código procedente de `/usr/libexec/ld.so` (¡Parte del enlazador dinámico!) y código procedente de la librería de C, procedente del fichero `/usr/lib/libc.so.78.1`.

Por el momento basta como introducción a lo que hace un sistema operativo y a qué relación tiene con tus programas. Ha llegado el momento de empezar a ver las abstracciones que implementa el kernel y las llamadas al sistema y comandos de shell relacionados con cada una de ellas. Empezaremos en el siguiente tema viendo lo que es un *proceso*, o programa en ejecución, lo que en realidad continúa lo que terminados de describir en este epígrafe.

Referencias

1. The C programming language, 2nd. ed. Brian W. Kernighan,Dennis M. Ritchie. Prentice Hall. 1988.
2. The UNIX Programming Environment. Brian W. Kernighan,Rob Pike. Prentice-Hall. 1984.
3. Operating Systems Design and Implementation. Andrew S. Tanembaum. PRHALL. 2004.
4. Commentary on UNIX 6th Edition, with Source Code. John Lions. Peer-to-Peer Communications. 1996.
5. The Design and Implementation of the 4.4 BSD Operating System. Marshall Kirk McKusick,Keith Bostic,Michael J. Karels,John S. Quarterman. Addison-Wesley. 1996.
6. Advanced Programming in the Unix Environment. Stevens. Addison-Wesley. 1992.
7. The Practice of Programming. Brian W. Kernighan,Rob Pike. Addison-Wesley. 1999.