# Introducción a Sistemas Operativos: Empezando

*Clips 6 a 10*
*Francisco J Ballesteros*

## 1. 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 1.

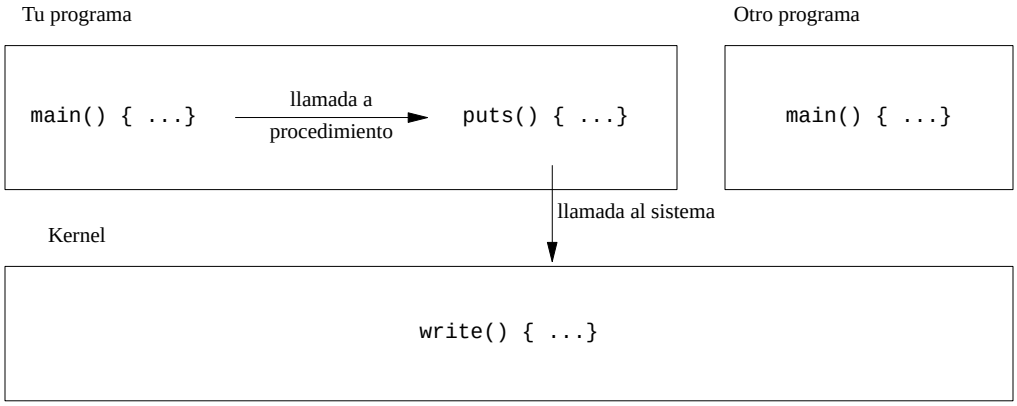


**Figura 1:** *Una llamada al sistema, utilizada por nuestro programa para saludar.*

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!

## 2. 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.