Introducción a Sistemas Operativos utilizando UNIX

Francisco J Ballesteros

(c) 2016 Francisco J. Ballesteros

Table of Contents

Capítulo 1: Empezando . . . 1
1.1 ¿Qué es el Sistema Operativo? . . . 2
1.2 ¿Qué es UNIX? . . . 4
1.3 ¿Cómo seguir este curso? . . . 5
1.4 Llamadas al sistema . . . 6
1.5 Comandos, Shell y llamadas al sistema. . . . 7
1.6 Obteniendo ayuda . . . 9
1.7 Utilizando ficheros . . . 12
1.8 Directorios . . . 16
1.9 ¿Qué contienen los ficheros? . . . 20
1.10 Usuarios y permisos . . . 22
1.11 Compilado y enlazado . . . 25
Capítulo 2: Procesos . . . 30
2.1 Programas y procesos . . . 31
2.2 Planificación y estados de planificación . . . 33
2.3 Carga de programas . . . 34
2.4 Nacimiento . . . 39
2.5 Muerte . . . 42
2.6 En la salida . . . 43
2.7 Errores . . . 44
2.8 Variables de entorno . . . 46
2.9 Procesos y nombres . . . 52
2.10 Usuarios . . . 53
2.11 ¿Qué más tiene un proceso? . . . 56
2.12 Cuando las cosas se tuercen... . . . 56
2.13 /proc . . . 60
Capítulo 3: Ficheros . . . 64
3.1 Entrada/Salida . . . 65
3.2 Open, close, el terminal y la consola . . . 70
3.3 Ficheros abiertos . . . 72
3.4 Permisos y control de acceso . . . 74
3.5 Offsets . . . 75
3.6 Ajustando el offset . . . 78
3.7 Crear y borrar . . . 80
3.8 Enlaces . . . 83
3.9 Lectura de directorios . . . 85
3.10 Globbing . . . 88
3.11 Metadatos . . . 91
3.12 El tiempo . . . 94
3.13 Cambiando los metadatos . . . 97
3.14 Enlaces simbólicos . . . 99
3.15 Dispositivos . . . 101
3.16 Entrada/salida y buffering . . . 103
3.17 Buffering en el kernel . . . 106
3.18 Ficheros proyectados en memoria. . . . 107
3.19 Montajes y nombres . . . 110
Capítulo 4: Padres e hijos . . . 113
4.1 Ejecutando un nuevo programa . . . 114
4.2 Creación de procesos . . . 115
4.2.1 Las variables . . . 117

(c) 2016 Francisco J. Ballesteros

4.2.2 El efecto de las cachés . . . 119
4.3 Juegos . . . 120
4.4 ¿Compartidos o no? . . . 121
4.4.1 Condiciones de carrera . . . 124
4.5 Cargando un nuevo programa . . . 124
4.6 Todo junto . . . 127
4.7 Esperando a un proceso hijo . . . 128
4.7.1 Zombies . . . 129
4.8 Ejecución en background . . . 129
4.9 Ejecutables . . . 130
4.9.1 Binarios . . . 130
4.9.2 Programas interpretados . . . 132
4.9.3 Scripts de shell . . . 135
Capítulo 5: Comunicación entre Procesos . . . 138
5.1 Redirecciones de Entrada/Salida . . . 139
5.2 Pipelines . . . 144
5.3 Juegos con pipes . . . 145
5.4 Pipeto . . . 149
5.5 Pipefroms . . . 151
5.6 Sustitución de comandos . . . 154
5.7 Pipes con nombre . . . 156
5.8 Señales . . . 159
5.9 Alarmas . . . 163
5.10 Terminales y sesiones . . . 165
5.10.1 Modo crudo y cocinado . . . 167
Capítulo 6: Usando el shell . . . 168
6.1 Comandos como herramientas . . . 169
6.2 Convenios . . . 169
6.3 Usando expresiones regulares . . . 171
6.4 Líneas y campos . . . 175
6.5 Funciones y otras estructuras de control . . . 180
6.6 Editando streams . . . 185
6.7 Trabajando con tablas . . . 189
Capítulo 7: Concurrencia . . . 193
7.1 Threads y procesos . . . 194
7.2 Condiciones de carrera . . . 196
7.3 Cierres . . . 200
7.4 Cierres en ficheros . . . 205
7.5 Cierres de lectura/escritura . . . 208
7.6 Cierres de lectura/escritura para threads . . . 210
7.7 Deadlocks . . . 211
7.8 Semáforos . . . 212
7.9 Semáforos en UNIX . . . 213
7.10 ¿Y si algo falla? . . . 215
7.11 Semáforos con pipes . . . 216
7.12 Buffers compartidos: el productor/consumidor . . . 219
7.13 Monitores . . . 224
7.14 Variables condición . . . 228
Capítulo 8: La red . . . 235
8.1 Sockets . . . 236
8.2 Un cliente . . . 236
8.3 Un servidor . . . 239
8.4 Usando el cliente y el servidor . . . 242

(c) 2016 Francisco J. Ballesteros

Capítulo 1: Empezando

(c) 2016 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.

(c) 2016 Francisco J. Ballesteros

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

(c) 2016 Francisco J. Ballesteros

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).

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.

(c) 2016 Francisco J. Ballesteros

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

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.

(c) 2016 Francisco J. Ballesteros

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.

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

(c) 2016 Francisco J. Ballesteros

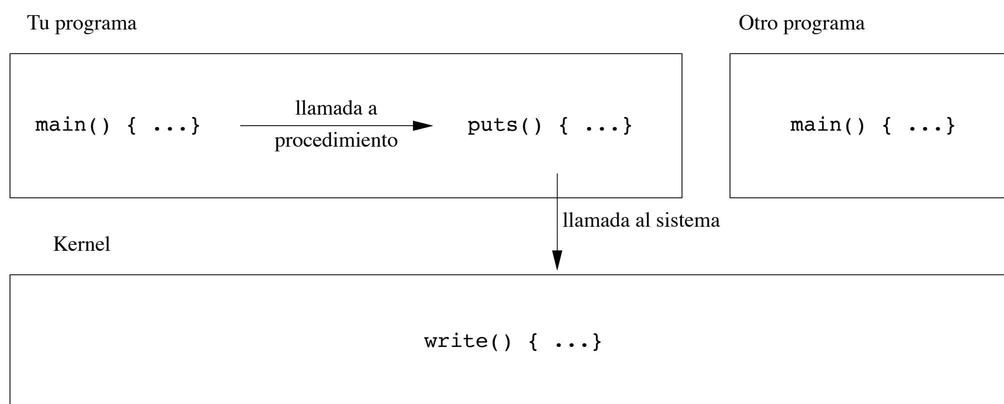

Figura 2: *Una llamada al sistema, utilizada por nuestro programa para saludar.*

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

(c) 2016 Francisco J. Ballesteros

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.

(c) 2016 Francisco J. Ballesteros

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:

(c) 2016 Francisco J. Ballesteros

```
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.

(c) 2016 Francisco J. Ballesteros

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:

(c) 2016 Francisco J. Ballesteros

```
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:

(c) 2016 Francisco J. Ballesteros

```
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:

(c) 2016 Francisco J. Ballesteros

```
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

(c) 2016 Francisco J. Ballesteros

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 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.

(c) 2016 Francisco J. Ballesteros

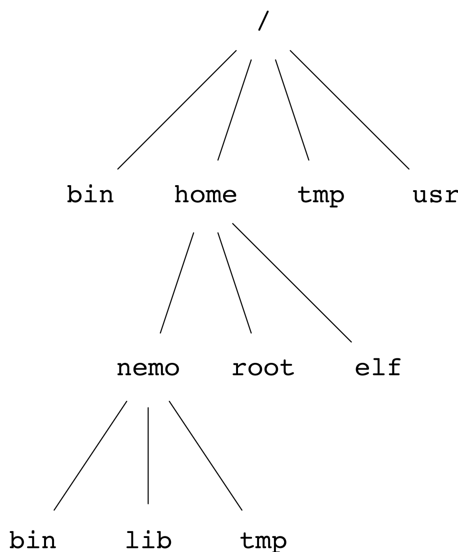

Figura 3: *Ejemplo de árbol de ficheros en UNIX.*

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.

(c) 2016 Francisco J. Ballesteros

```
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
```

(c) 2016 Francisco J. Ballesteros

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 .
```

(c) 2016 Francisco J. Ballesteros

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.

(c) 2016 Francisco J. Ballesteros

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:

(c) 2016 Francisco J. Ballesteros

```
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 *codepoint* 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.

(c) 2016 Francisco J. Ballesteros

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,

(c) 2016 Francisco J. Ballesteros

```
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:

(c) 2016 Francisco J. Ballesteros

```
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.

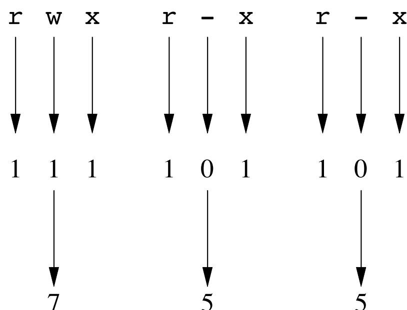

Figura 4: *Los permisos en UNIX se codifican como un número en octal en muchas ocasiones.*

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");
}
```

(c) 2016 Francisco J. Ballesteros

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

(c) 2016 Francisco J. Ballesteros

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

(c) 2016 Francisco J. Ballesteros

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:

(c) 2016 Francisco J. Ballesteros

```
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.

(c) 2016 Francisco J. Ballesteros

Capítulo 2: Procesos

(c) 2016 Francisco J. Ballesteros

1. Programas y procesos

A un programa en ejecución se lo denomina *proceso*. El nombre *programa* no se utiliza para referirse a un programa que está ejecutando dado que ambos conceptos (programa y proceso) son diferentes. La diferencia es la misma que hay entre una receta de cocina para hacer galletas y una galleta hecha con la receta. El programa es únicamente una serie de datos (con las instrucciones y datos para ejecutarlo) y no es algo vivo. Por otro lado, un proceso es un programa que está vivo. Tiene una serie de registros, incluyendo un contador de programa, y utiliza una pila para hacer llamadas a procedimiento (y al sistema). Esto significa que un proceso tiene un *flujo de control* que ejecuta una instrucción tras otra, como ya sabes.

La diferencia queda clara si consideras que puedes ejecutar a la vez el mismo programa, varias veces. Por ejemplo, la figura 5 muestra varias ventanas que están ejecutando un shell.

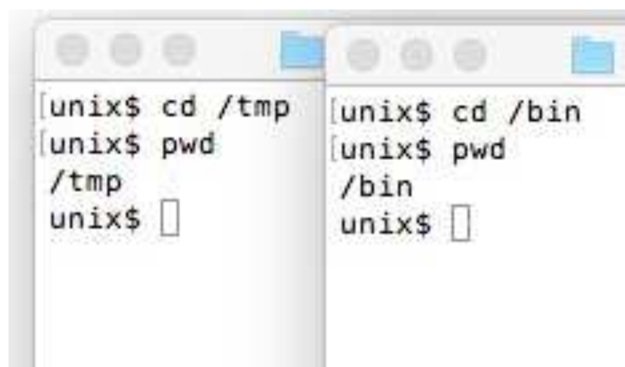

Figura 5: *Varias ventanas ejecutando un shell. Hay solo un programa pero tenemos varios procesos.*

En ambos casos ejecutan el mismo programa: el que está en `/bin/sh`. Pero, aunque hay un sólo programa, tenemos varios procesos. Por ejemplo, si cambiamos de directorio en uno de ellos, el otro sigue inalterado. Cada proceso tiene sus propias variables en la memoria. Y, aunque el programa es el mismo, los valores de las variables serán en general diferentes en cada proceso. Esto es obvio si piensas que cada shell estará haciendo una cosa distinta en un momento dado. No obstante, el programa tiene únicamente un conjunto de variables declaradas.

¿Qué es entonces un proceso? Piensa en los programas que has hecho. Elige uno de ellos. Cuando lo ejecutas, comienza a ejecutar *independientemente* del resto de programas en el ordenador. Al programarlo, ¿has tenido que tener en cuenta otros programas como el shell, el sistema de ventanas, el reloj, el navegador web, ...? ¡Por supuesto que no! Haría falta un cerebro del tamaño de la luna para tener todo eso en cuenta. Dado que no existen esos cerebros, el Sistema Operativo se ocupa de darte como abstracción el *proceso*. Gracias a esa abstracción puedes programar y ejecutar un programa olvidando el resto de programas que tienes en el sistema: Los procesos son programas que ejecutan independientemente del resto de programas que están ejecutando en el sistema.

Cada proceso viene con la ilusión de tener su propio procesador. Naturalmente, esto es mentira y es parte de la abstracción. Cuando escribes un programa piensas que la máquina ejecuta una instrucción tras otra. Y piensas que toda la máquina es tuya. Bueno, en realidad, que toda la máquina es del programa que ejecuta. La implementación de la abstracción *proceso* es responsable de esta fantasía.

Cuando existen varios procesadores o CPUs, varios programas pueden ejecutarse realmente a la vez, o *en paralelo*. Hoy en día, la mayoría de los ordenadores disponen de múltiples procesadores. Pero, en cualquier caso, seguramente ejecutes más programas que procesadores tienes. ¡Al mismo tiempo! Cuenta el número de ventanas que tienes abiertas y piensa que hay al menos un programa ejecutando por cada ventana. Seguro que no tienes tantos procesadores.

Lo que sucede es que el sistema hace los ajustes necesarios para que cada programa pueda ejecutar durante un tiempo. La figura 6 muestra la memoria del sistema con varios procesos ejecutando. Cada uno tiene su

(c) 2016 Francisco J. Ballesteros

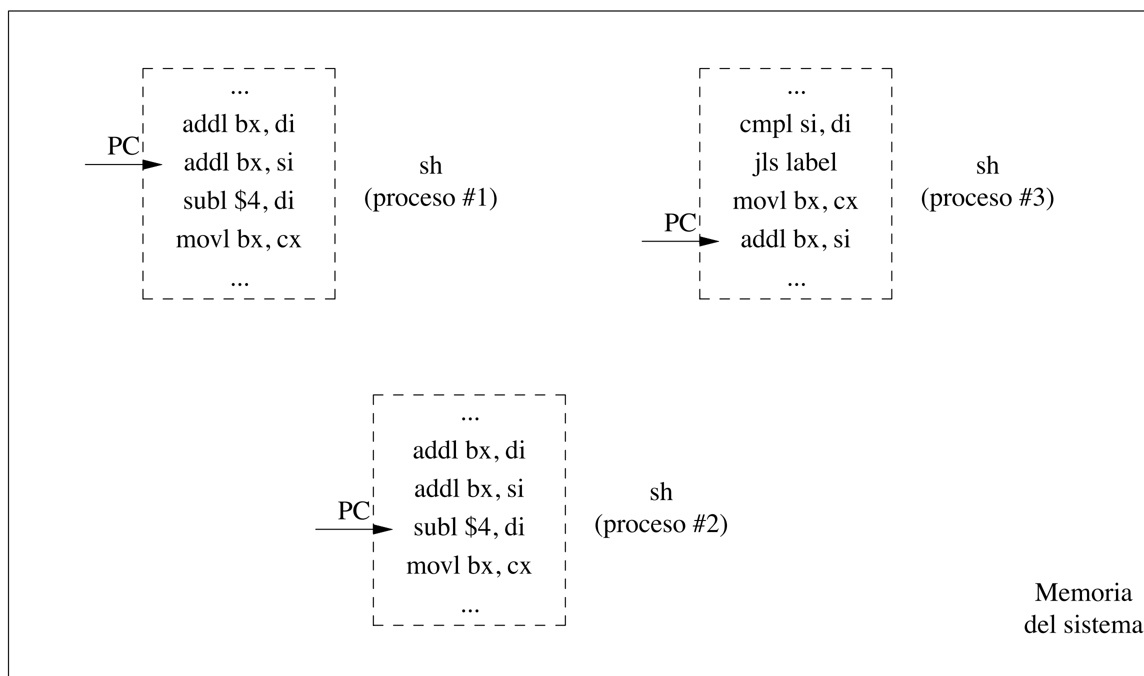

Figura 6: *Ejecución concurrente de varios procesos en el mismo sistema.*

propio juego de registros, incluyendo el contador de programa. Además, cada uno tendrá su propia pila. La figura es tan solo una imagen de la memoria en un momento dado. Durante algún tiempo, el proceso 1 que ejecuta `sh` estará ejecutando en el procesador (sin que el Sistema Operativo intervenga mientras tanto). Después, un temporizador hardware arrancado anteriormente por el sistema interrumpirá la ejecución de dicho programa. En ese momento el sistema puede decidir que el proceso 1 ya ha tenido suficiente tiempo de ejecución por el momento, y en tal caso puede *saltar* para continuar ejecutando (por ejemplo), el proceso 2. Este proceso puede estar también ejecutando `sh`, o cualquier otro programa. Pasado cierto tiempo, otra interrupción de reloj llegará y una vez más se interrumpirá la ejecución del proceso que ejecuta. En este caso, el sistema (donde está el código del programa que atiende la interrupción) puede decidir que ya es hora de volver a ejecutar el proceso 1. De ser así, saltará al punto por el que estaba ejecutando dicho proceso la última vez que ejecutó.

Todo esto sucede detrás del telón. El sistema sabe que hay tan solo un flujo de control por cada procesador, y salta de un programa a otro utilizando dicho flujo. Pero, para los usuarios del sistema, lo que importa es que cada proceso parece estar ejecutando de forma independiente del resto. ¡Como si tuviera un procesador dedicado para el solo!

Como todos los procesos aparentan ejecutar de forma simultánea, decimos que ejecutan *concurrentemente*. O dicho de otro modo, decimos que son procesos concurrentes. En algunos casos, realmente ejecutan a la vez (cada uno en un procesador). De ser así, decimos que ejecutan *paralelamente*. En la mayoría de los casos compartirán procesador (un tiempo cada uno) y diremos que son *pseudo-paralelos*. Pero, en la práctica, da igual si ejecutan de un modo u otro y simplemente los consideramos concurrentes. A la hora de programar y de utilizar el sistema esto es todo lo que importa.

En este tema vamos a explorar el proceso que obtenemos cuando ejecutamos un programa. Pero antes de hacer esto es importante ver qué hay en un programa y qué hay en un proceso, cosa que haremos a continuación.

(c) 2016 Francisco J. Ballesteros

2. Planificación y estados de planificación

Supongamos que hay sólo un procesador. De haber varios, lo que vamos a decir se aplica a cada uno de ellos sin más complicación. Cada proceso aparenta ejecutar de forma independiente, con su flujo de control propio, aunque el procesador sólo implementa un flujo de control.

Lo que sucede es que cuando un proceso entra al kernel, ya sea por una interrupción procedente del hardware o por que hace una llamada al sistema, el sistema tiene almacenado el estado del proceso que estaba ejecutando (normalmente en la pila en que se ha salvado el estado en la interrupción o al entrar al sistema). Gracias a esto, el sistema puede decidir que hay que saltar hacia otro proceso y puede *retornar* de la interrupción o llamada al sistema utilizando el estado que se salvó anteriormente para *otro* proceso distinto. Por ejemplo, se puede cambiar la pila por la de otro proceso de tal forma que cuando se retorne sea hacia ese otro proceso. De este modo, a cada proceso se le da un tiempo de procesador y, después, el sistema salta a otro distinto. A esta cantidad de tiempo se la denomina *cuanto*, y puede ser del orden de 100ms (lo que es una enormidad de tiempo para la máquina).

A transferir el control de un proceso a otro, salvando el estado del antiguo y recargando el estado del nuevo, lo denominamos *cambio de contexto*. Esto es obvio si pensamos que cambiamos el *contexto* (registros, pila, etc.) de un proceso a otro. Es de notar que es el kernel el que hace estos cambios de contexto. ¡Tu nunca incluyes saltos en tus programas para que otros programas ejecuten!

La parte del kernel responsable de decidir qué proceso es el siguiente que ejecuta en un cambio de contexto se denomina *planificador*, o *scheduler*. Es fácil si pensamos en que planifica la ejecución de los procesos. A las decisiones del planificador las conocemos como *planificación*, o *scheduling*. ¡Sorprendente! En la mayoría de sistemas el kernel puede sacar del procesador a un proceso incluso si este no hace llamadas al sistema (por ejemplo, mientras ejecuta un bucle). Se suelen utilizar las interrupciones de reloj a tal efecto. En este tipo de planificación, decimos que tenemos un planificador expulsivo (o, en Inglés, *preemptive*). Cuando el planificador no utiliza interrupciones para expulsar procesos decimos que tenemos un planificador *cooperativo*. Pero, en este último caso, un programa que entre en un bucle infinito sin llamar al sistema puede convertir la máquina en un pisapapeles.

Con un único procesador, sólo un proceso puede ejecutar en cada momento. El resto en general pueden estar *listos para ejecutar* (*ready*) pero no estarán ejecutando. Ya conoces dos estados de planificación: *listo* y *ejecutando*. Un proceso ejecutando pasa a estar listo para ejecutar si lo expulsan del procesador cuando su tiempo termina. En ese momento, el kernel ha de elegir a un proceso de entre los que están listos para ejecutar y ha de ponerlo a ejecutar.

Los estados de planificación son tan sólo constantes que define el sistema para guardarlas en los *records* o *structs* con los que implementa los procesos y saber así si puede elegir un proceso para ejecutarlo o no. La figura 7 muestra los estados de planificación de un proceso.

En algunos casos un proceso leerá de teclado o de una conexión de red o cualquier otro dispositivo. Cuando esto ocurre, el proceso habrá de esperar a que el usuario pulse una tecla, a que lleguen datos, o a que suceda alguna otra cosa. El proceso podría esperar entrando en un bucle, pero eso sería un desperdicio de procesador. En lugar de eso, cuando un proceso llama al sistema y el kernel ve que no puede continuar, se le marca como *bloqueado* (otro estado de planificación) para que el kernel no lo ponga a ejecutar.

Los dispositivos en entrada/salida son tan lentos en comparación con el procesador que es posible ejecutar multitud de cosas mientras un proceso espera a que su entrada/salida termine. A hacer tal cosa se la denomina *multiprogramación*.

Pasado cierto tiempo, un proceso bloqueado volverá a marcarse como listo para ejecutar si el evento que estaba esperando el proceso sucede. Por ejemplo, si el proceso llamó al kernel para leer de teclado y, debido a eso, se bloqueó, en el momento en que escribimos algo en el teclado y el kernel puede dárselo el proceso,

(c) 2016 Francisco J. Ballesteros

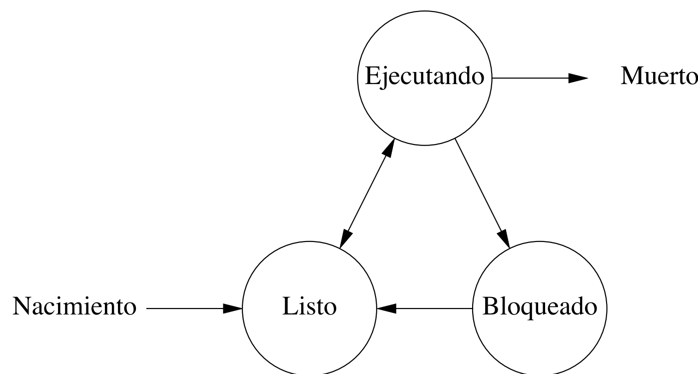

Figura 7: *Estados de planificación de un proceso.*

dicho proceso pasará a estar listo para ejecutar. La siguiente vez que el planificador ejecute, el proceso es un candidato a ejecutar.

Otra forma de dibujar el estado de los procesos a lo largo del tiempo es utilizar un *diagrama de planificación*. Se trata de dibujar una línea para cada proceso de tal forma que indicamos en qué estado está en cada momento (cambiando la forma o el color de la línea). Naturalmente, si tenemos un procesador, sólo podrá existir un proceso ejecutando en cada momento. Y además, ya sabes que en realidad el salto de un proceso a otro sucede utilizando el mecanismo que hemos descrito antes. Un ejemplo de diagrama de planificación podría ser el que puede verse en la figura 8. En dicha figura, hay dos procesos ejecutando concurrentemente en un sólo procesador. Uno ejecuta `sh` y el otro `ls`.

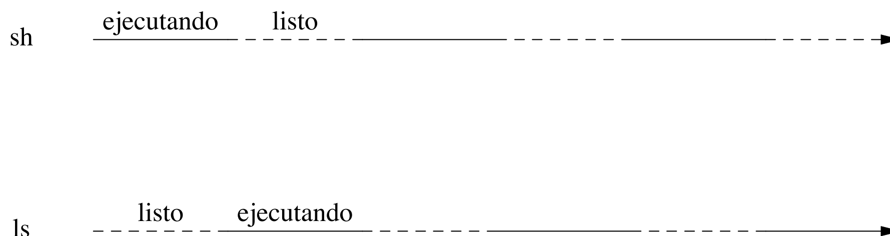

Figura 8: *Diagrama de planificación. El tiempo fluye hacia la derecha en la figura.*

3. Carga de programas

Vimos que cuando un programa se compila y se enlaza obtenemos un fichero ejecutable. Este fichero contiene toda la información necesaria para que el Sistema Operativo pueda ponerlo a ejecutar (convertirlo en un proceso). Hay diferentes partes del binario que contienen tipos diferentes de información (código, datos, etc.) y se denominan *secciones* del ejecutable. Además, el ejecutable suele comenzar con una estructura de datos denominada cabecera que incluye información sobre qué secciones están presentes en el ejecutable. Concretamente, dónde empiezan en el fichero, de qué tipo son y qué tamaño tienen.

Una sección del fichero contiene el texto del programa (el código). Las variables globales inicializadas están en otra sección. O, con más precisión, los bytes que corresponden a los datos inicializados están en dicha sección. En realidad el sistema no sabe *nada* respecto a qué variables hay en tu programa o qué significan dichas variables. Simplemente se guardan en el ejecutable los bytes que, una vez cargados en la memoria, dan el valor inicial a las variables que utilizas. Para variables sin inicializar, sólo es preciso guardar en la cabecera del fichero ejecutable qué tamaño en bytes es preciso para dichas variables (y a partir de qué dirección estarán cargadas en la memoria). Dado que no tienen valor inicial, o mejor dicho, dado

(c) 2016 Francisco J. Ballesteros

que su valor inicial es cero (todos los bytes a cero), no es preciso guardar los ceros en el ejecutable. Con indicar el tamaño de dicha sección de memoria basta.

Habitualmente, el ejecutable contiene también una tabla de símbolos con información para depuradores y programas como *nm(1)* que indica los nombres de los símbolos en el fuente, y los nombres de los ficheros fuente y números de línea. Esta información no es para el sistema operativo, es para ti y para el depurador. Para el sistema, tu programa no tiene ningún significado en particular. Sólo tu código sabe el significado de los bytes en los datos que manipula (si son enteros que hay que sumar, o qué hay que hacer con ellos).

Ya vimos como utilizar *nm* para mostrar la información de ficheros objeto y ejecutables. Consideremos ahora el programa del siguiente listado.

[global.c]:

```
#include <stdlib.h>

char global[1 * 1024 * 1024];
int global2;
int init1 = 4;
int init2 = 3;
static int stint1;
static int stint2 = 3;

int
main(int argc, char*argv[])
{
        exit(0);
}
```

Este programa tiene declaradas varias variables globales. Por un lado, `global` y `global2` son variables globales sin inicializar. Igualmente, `stint1` es una global sin inicializar, aunque declarada *static* (con lo que no se exporta desde el fichero objeto y sólo puede usarse desde el fichero que estamos compilando). Además, tenemos variables globales llamadas `init1`, `init2` y `stinit2` que están inicializadas. Si utilizamos *nm* en el ejecutable resultante de compilar y enlazar este programa vemos lo siguiente:

(c) 2016 Francisco J. Ballesteros

```
unix$ cc global.c
unix$ nm -n a.out
00000a30 T __start
00000a30 T _start
00000c30 T atexit
00000ca4 T main
...
00201008 D __progname
00201010 D __dso_handle
00201020 D init1
00201024 D init2
00201028 d stinit2
...
00401320 A __bss_start
00401320 A _edata
00401360 b stint1
00401380 B _dl_skipnum
004014a0 B global2
004014c0 B global
005014c0 A _end
crun%
```

El comando *nm* nos informa de las direcciones en que podremos encontrar, una vez cargado en la memoria, cada uno de los símbolos del ejecutable. Como puedes vers, el código del texto del programa (las instrucciones) estarán a partir de la dirección `a30` de memoria. Esos símbolos se muestran con la letra "`T`" en la salida de *nm* para indicar que son de texto.

Las variables `init1`, `init2`, y `stinit2` estarán a partir de la dirección `201020` (en hexadecimal). Puedes ver como cuatro bytes después de `init1`, en la dirección `201024`, estará `init2`. Dado que `init1` es un *int* y que en este sistema (y con este compilador) ocupa 4 bytes, eso tiene sentido. Todas estos símbolos se muestran con la letra "`D`" (o "`d`"), indicado que corresponden a datos inicializados. Las letras que muestra *nm* son mayúsculas cuando los símbolos se exportan desde el fichero objeto (pueden usarse desde otros ficheros objeto dentro del mismo ejecutable) y minúscula en caso contrario. Fíjate en `init2` y en `stinit2` en la salida de *nm* y en el código fuente. ¿Ves la diferencia?

Las variables `global2` y `global` se muestran con la letra "`B`", indicando que corresponden a variables sin inicializar. Para estas variables, el valor inicial será cero (todos los bytes a cero). Puedes ver cómo la variable `global` está en la dirección `004014c0` y, un Mbyte después, en la dirección `005014c0` está la siguiente variable.

El código ocupará una zona de memoria contigua y tendrá permisos para permitir la ejecución de instrucciones. Habitualmente, no tendrá permisos para permitir la escritura de esa zona de memoria. El contenido de esta memoria se inicializa leyendo los bytes del fichero ejecutable que corresponden al texto del programa. Los datos inicializados ocuparán otra zona de memoria contigua, con permisos para leer y escribir dicha memoria. Esta memoria se inicializará leyendo del fichero ejecutable los bytes que corresponden al valor inicial de esa zona de memoria. Los datos sin inicializar ocuparán otra zona, inicializada a cero.

Cada una de estas zonas es contigua, tiene una dirección de comienzo y un tamaño, y la memoria que ocupa tiene las mismas propiedades (permisos, cómo se inicializa, etc.). Es precisamente a eso a lo que se denomina *segmento*. Cuando el program ejecute, tendrá un segmento de texto con el código, otro de datos con variables inicializadas, otro denominado *BSS* con datos sin inicializar (que parten con los bytes a cero) y otro de pila. Véase la figura 9. El nombre BSS viene de una instrucción de una máquina que ya no existe que se utilizaba para inicializar memoria a cero (y son las iniciales de *Base Stack Segment*, aunque hoy día no tiene que ver con la pila).

(c) 2016 Francisco J. Ballesteros

Cuando el sistema ejecute el programa contenido en este fichero ejecutable, cargará en la memoria el código procedente del mismo, y los valores iniciales para variables inicializadas. Igualmente, la memoria que ocupen las variables sin inicializar se inicializará a cero y se creará una pila para que puedan hacerse llamadas a procedimiento. El trozo de código del kernel que se ocupa de todo esto se denomina cargador.

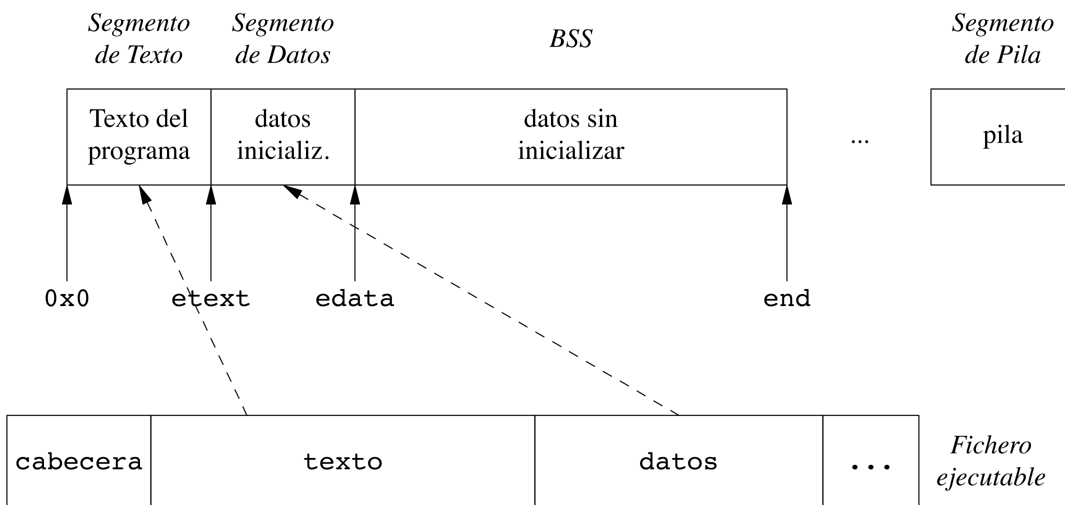

Figura 9: *El cargador se ocupa de cargar los segmentos de un proceso con datos procedentes del ejecutable.*

Pero es importante que comprendas que el sistema *no sabe nada* sobre tu programa. Ni siquiera sabe ni qué funciones has definido ni qué variables utilizas. ¡Ni le importa! Si utilizamos el comando *strip(1)* para eliminar la información de la tabla de símbolos del ejecutable, podremos verlo.

```
unix$ ls -l a.out
-rwxr-xr-x  1 nemo  wheel  8729 May 23 14:59 a.out*
unix$ strip a.out
unix$ ls -l a.out
-rwxr-xr-x  1 nemo  wheel  6776 May 23 15:26 a.out*

unix$ nm -n a.out
00000000 W __cxa_atexit
00000cc0 T __fini
00201008 D __data_start
00201008 D __progname
003012a0 D __got_start
00301320 D __got_end
00401320 A __bss_start
00401320 A _edata
005014c0 A _end
```

Tras ejecutar *strip*, *nm* no puede obtener información alguna respecto que qué variables globales o qué funciones están contenidas en el ejecutable. Como hemos compilado con enlace dinámico aún quedan algunos símbolos que son necesarios para que el enlazado dinámico funcione, pero eso es todo. ¡Nadie puede saber que había algo llamado `global` en tu código!

El cargador se limitará a cargar los bytes de texto hacia el segmento de texto y los bytes de datos inicializados hacia el segmento de datos. Lo que signifiquen esos bytes es cosa tuya. En cuanto al BSS, el ejecutable indica en la cabecera cuantos bytes son necesarios, pero naturalmente no guardamos 1Mbyte de ceros en el ejecutable para nuestra variable `global`. Simplemente se indica en el ejecutable en número total de bytes a cero y la dirección en que deben comenzar.

(c) 2016 Francisco J. Ballesteros

Volvamos a pensar en los segmentos de un proceso. En la figura 9 puedes ver que el enlazador suele dejar en el ejecutable un símbolo llamado `etext` cuya dirección es el final de texto, otro `edata` cuya dirección es el final de los datos inicializados y otro `end` cuya dirección es el final del BSS. Con estos símbolos puedes saber qué direcciones utilizan tus principales segmentos. El comando *size(1)* muestra el tamaño que tendrán los segmentos de un programa cuando lo ejecutes:

```
unix$ size a.out
text    data    bss     dec     hex
2845    500     1048992 1052337 100eb1
```

Para nuestro programa, el BSS tendrá `1048992` bytes.

Los segmentos son parte de la abstracción que te da UNIX para que tengas procesos. Como habrás observado, aunque ejecutemos muchos procesos, todos ellos piensan que tienen la memoria para ellos solos. Por ejemplo, en nuestro programa la variable `global` estaría en la dirección `004014c0`. Si lo ejecutamos varias veces de forma simultánea, cada uno de los procesos tendrá su propia versión y pensará que es el único en el mundo. En todos ellos, `global` estará en la dirección `004014c0`.

Simplemente, la memoria del proceso es memoria virtual. Esta memoria no existe en realidad. Por eso se la denomina virtual. El sistema utiliza el hardware de traducción de direcciones (la unidad de gestión de memoria o MMU) para hacer que las direcciones que utiliza el procesador se cambien por otras *al vuelo* antes de que el hardware de memoria las vea. Cada proceso utilizará zonas distintas de memoria física y, gracias a la traducción de direcciones, la dirección `004014c0` de cada proceso se cambia por la dirección de memoria física que dicho proceso utilice. Pero todos ejecutan instrucciones que acceden a `global` a partir de la dirección `004014c0`.

Las direcciones de memoria virtual se traducen a direcciones de memoria física empleando una tabla (que inicializa para cada proceso el sistema operativo) llamada *tabla de páginas*. Esta tabla traduce de 4 en 4 Kbytes (o un tamaño similar) las direcciones. A cada trozo de 4 Kbytes de memoria virtual lo llamamos página y al trozo correspondiente de memoria física lo denominamos marco de página.

La primera página de memoria suele dejarse sin traducción, lo que hace que sea imposible utilizar sus direcciones sin ocasionar un error (similar al que se produciría si divides por cero). Este error o excepción se atiende de forma similar a una interrupción y, habitualmente, el sistema operativo detiene la ejecución del programa que lo ha causado. Por eso no puedes atravesar un puntero a *nil*. La dirección de memoria cero es inválida, y *nil* es el valor cero utilizado como dirección de memoria.

Todo esto es importante puesto que tiene impacto en tus programas. Habitualmente el sistema carga tu código y datos, y te asigna memoria física, conforme utilizas direcciones de memoria virtual en tus segmentos. Inicialmente no habrá traducciones a memoria física para casi nada en tu proceso y, conforme el procesador utilice direcciones, el sistema irá cargando en demanda el resto. A esto se lo denomina *paginación en demanda*.

La paginación en demanda es fácil de implementar puesto que el sistema sabe qué segmentos tiene el proceso y qué direcciones usan estos. Además, sabe cómo inicializar la memoria de dichos segmentos (leyendo valores iniciales del fichero ejecutable o inicializándola a cero, por ejemplo). Inicialmente, un segmento puede tener sus páginas sin traducción a (marcos de página en) memoria física. Cuando el procesador utiliza una dirección en dichas páginas, el hardware eleva una excepción y el manejador instalado por el sistema operativo la atiende. Dicho manejador comprueba que en teoría el proceso debería poder utilizar dicha página de memoria y asigna un marco de página (poniendo una traducción en la tabla de páginas) inicializado según corresponda. Cuando el manejador retorna, el programa de usuario continúa su ejecución como si nunca hubiese sucedido el fallo de página.

Por ejemplo, `global` está inicializado a cero porque las páginas en que reside serán inicializadas a cero por el sistema cuando las asigne. Y será así dado que dichas páginas forman parte del BSS. Si tu proceso

(c) 2016 Francisco J. Ballesteros

lee una posición de `global` por primera vez, es muy posible que al proceso se le asignen 4Kbytes de memoria física para la página a que accede, inicializados a cero. Y por eso tu proceso creerá que `global` está inicializado a cero. Antes de utilizar el array, el MByte es sólo memoria virtual. ¡No consume memoria física! Eso si, si se te ocurre la "buena" idea de utilizar un bucle para inicializar tu array a cero, tu proceso accederá a todas esas direcciones y el sistema le asignará la memoria correspondiente. La memoria virtual es gratis, la memoria física no. Es muy habitual utilizar arrays de gran tamaño sabiendo que sólo van a utilizarse unas pocas posiciones en tiempo de ejecución, por ejemplo para grandes tablas hash y para otros propósitos en algunos run-times de lenguajes de programación. ¿Comprendes por qué puede hacerse esto?

Sucede lo mismo con la pila hoy día. Hace tiempo, el segmento de pila solía crecer a medida que el proceso utilizaba más espacio en la misma. Hoy día, el segmento de pila suele tener un tamaño prefijado y no crece. Dado que la memoria que usa es *virtual*, inicialmente dicha memoria es gratis: no tiene asignada memoria física. Naturalmente, esto es así hasta que dicha memoria se usa a medida que crece la pila.

4. Nacimiento

A los subprogramas se los llama y retornan, pero si consideramos un programa que queremos ejecutar en un proceso, no hay *llamadas* a programas ni dichos programas *retornan*. Un proceso simplemente termina cuando le pide al sistema terminar o cuando se comporta mal (por ej., intenta leer una dirección de memoria que no está en ningún segmento) y el sistema lo mata. No obstante, como sabes, cuando ejecutamos comandos en el shell podemos indicar argumentos para que sus programas los procesen y para controlar lo que hacen.

Cuando el shell el pide al sistema que ejecute un programa, una vez que dicho programa está cargado en memoria, el sistema suministra un flujo de control para que ejecute. En realidad sabes que lo habitual es que *no* se cargue todo el programa en memoria y, en cambio, se pagina en demanda aquellas páginas de memoria a las que accede el programa conforme ejecuta. Pues bien, el flujo de control supone valores para los registros del procesador, inicializados para que comiencen la ejecución del programa que se desea ejecutar e incluyendo un contador de programa y un puntero de pila. La pila inicialmente estará prácticamente vacía salvo por los argumentos del programa principal. Cuando compilamos un programa en C, el enlazador se ocupa de indicar en el ejecutable que la dirección de `main` es la dirección en la que hay que empezar a ejecutar código. Es por eso que los programas en C comienzan ejecutando `main`. Los argumentos de `main` son un array de strings (en `argv`) y el número de strings que hay en dicho array (en `argc`).

Teniendo esto en cuenta, el siguiente programa escribe sus argumentos indicando antes de cada uno el índice en `argv`:

```
#include <stdlib.h>
#include <stdio.h>
int
main(int argc, char* argv[])
{
    int i;

    for(i = 0; i < argc; i++) {
        printf("%d\t%s\n", i, argv[i]);
    }
    exit(0);
}
```

Si lo guardamos en el fichero `eco.c` y lo compilamos y lo ejecutamos podemos ver qué argumentos recibe el programa para una línea de comandos dada:

(c) 2016 Francisco J. Ballesteros

```
unix$ cc eco.c
unix$ ./eco un programa sencillo
0:    ./eco
1:    un
2:    programa
3:    sencillo
unix$
```

Como verás, ¡el primer "argumento" en `argv` es en realidad el nombre del programa! Concretamente, es el nombre del programa tal y como lo hemos escrito en la línea de comandos del shell. Recuerda que "`./eco`" es un camino o path relativo y que "`.`" es el directorio actual. Así pues, `argv[0]` contiene el nombre del programa. Los siguientes strings en `argv` son los argumentos que hemos dado a `eco` en la línea de comandos. Es muy útil emplear `argv[0]` en mensajes de error para identificar ante el usuario al programa que tiene problemas.

Lo que ha sucedido es que el shell ha leído la línea de comandos que hemos escrito, y ha utilizado la primera palabra para localizar el fichero que contiene el programa que queremos ejecutar. Después, le ha pedido a UNIX que lo ejecute y que haga que `main` en dicho programa reciba una copia del array de strings que corresponde a la línea de comandos.

Ya conocemos *echo(1)*. Recuerda que dicho comando acepta la opción "`-n`" para suprimir la impresión del fin de línea tras imprimir sus argumentos. Podemos cambiar el programa anterior para que sea como *echo(1)*, y podemos hacer que la opción "`-v`" imprima cada argumento entre corchetes, para distinguirlos mejor. Este es el programa:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

static char *argv0;

static void
usage(void)
{
    fprintf(stderr, "usage: %s [-nv] [args...]\n", argv0);
    exit(1);
}
```

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    int i, nflag, vflag;
    char *arg, *sep;

    nflag = vflag = 0;
    argv0 = argv[0];
    argv++;
    argc--;
    while(argc > 0 && argv[0][0] == '-') {
        if (strcmp(argv[0], "--") == 0) {
            argv++;
            argc--;
            break;
        }
        for(arg = argv[0]+1; *arg != 0; arg++) {
            switch(*arg) {
            case 'n':
                nflag = 1;
                break;
            case 'v':
                vflag = 1;
                break;
            default:
                usage();
            }
        }
        argv++;
        argc--;
    }
    sep = "";
    for(i = 0; i < argc; i++){
        if (vflag) {
            printf("%s[%s]", sep, argv[i]);
        } else {
            printf("%s%s", sep, argv[i]);
        }
        sep = " ";
    }
    if (!nflag) {
        printf("\n");
    }
    exit(0);
}
```

Primero guardamos en la global `argv0` el nombre del programa. Esto lo usamos en la función `usage` para imprimir un recordatorio del uso del programa empleando el nombre tal y como lo hemos recibido en `argv[0]`. Una vez hecho esto, quitamos del vector del argumentos el primer string haciendo que `argv` apunte al siguiente elemento (y actualizando `argc` para que refleje el número de strings en `argv`).

El bucle `while` procesa las opciones (argumentos que empiezan por "`-`") para detectar si se ha indicado "`-n`" o "`-v`". Como verás, el programa también entiende dichas opciones si se suministra "`-nv`" o "`-vn`", como suelen hacer los programas en UNIX. Además, un argumento "`--`" indica el fin de las opciones, para

(c) 2016 Francisco J. Ballesteros

hacer que el resto de argumentos no se procesen como opciones. Esto es lo habitual en UNIX. Por ejemplo, podemos ejecutar el programa como sigue, si lo tenemos compilado y enlazado en el ejecutable "`eco2`".

```
unix$ eco2 -v -- -n hola
[-n] [hola]
unix$
```

Hemos optado por procesar los argumentos de tal forma que una vez procesadas todas las opciones, `argc` y `argv` contienen el resto de argumentos. Esto es cómodo en general.

Si llamamos al programa de forma incorrecta, obtenemos un recordatorio de cómo hay que usarlo:

```
unix$ eco2 -vx hola
usage: eco2 [-nv] [args...]
unix$
```

5. Muerte

Habrás notado que "`main`" termina llamando a `exit`. Esta llamada al sistema pide a UNIX que termine la ejecución del proceso (y por tanto de su programa). El entero que pasamos como argumento se espera que sea cero si el programa ha conseguido hacer su trabajo y distinto de cero en caso contrario. A este valor se le suele llamar **exit status** (en inglés). Todo esto es muy importante. Habitualmente es un programa el que ejecuta otros programas (por ejemplo el shell). Si dichos programas no informan correctamente respecto a si pudieron hacer su trabajo o no, el programa que los ejecuta podría hacer cosas que no esperamos.

Si un programa no comprueba si sus argumentos son correctos y/o no suministra el estatus de salida adecuado (cero o distinto de cero), no está correctamente programado y es posible que no pueda usarse en la práctica.

Habrás visto que `main` retorna un entero. Dado que *exit(3)* termina la ejecución, no retornamos nada en nuestro programa. No obstante, retornar de `main` hace que el valor retornado se utilice para llamar a *exit(3)* con dicho valor. Esto es, podríamos haber terminado nuestro programa escribiendo

```
int
main(int argc, char* argv[])
{
    ...
    return 0;
}
```

No hay magia en esto. En realidad el programa principal es una función que hace algo equivalente a

```
exit(main(argc, argv));
```

lo que explica que podamos retornar de `main`.

Podemos utilizar el shell para ver que tal le fue a un comando que ejecutamos anteriormente:

```
unix$ ls /blah
ls: /blah: No such file or directory
unix$ echo $?
1
unix$ echo $?
0
```

(c) 2016 Francisco J. Ballesteros

Aquí, "`$?`" es una *variable de entorno* (más adelante veremos lo que es esto) que contiene como valor un string correspondiente al estatus de salida del último comando ejecutado. ¿Puedes decir por qué la segunda vez el estatus es 0?

6. En la salida

En ocasiones resulta útil ejecutar código cuando un programa termina. Aunque no se debe abusar de este mecanismo, puede resultar útil para, por ejemplo, borrar ficheros temporales o realizar alguna otra tarea incluso si una función decide llamar a `exit`.

La función *atexit(3)* permite instalar punteros a función de tal forma que dichas funciones ejecuten durante la llamada a *exit(3)*. En realidad, `exit` llama las funciones que ha instalado `atexit` y luego llama a `_exit`, que realmente termina la ejecución del proceso.

Recuerda que el programa principal es el realidad

```
exit(main(argc, argv));
```

La función de librería `exit` se parece a...

```
void
exit(int sts)
{
    int i;
    for (i = 0; i < numexitfns; i++) {
        exitfns[i]();
    }
    _exit(sts);
}
```

Para ejecutar algo cuando el programa termine llamando a *exit*, podemos llamar a *atexit* como en el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>

static void
exitfn1(void)
{
    puts("dentro de exitfn1");
}

static void
exitfn2(void)
{
    puts("dentro de exitfn2");
}
```

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    atexit(exitfn1);
    atexit(exitfn2);
    puts("a punto de salir...");
    return 0;
}
```

Cuando lo ejecutemos, suponiendo que el ejecutable es `atexit`, veremos algo como...

```
unix$ atexit
a punto de salir...
dentro de exitfn2
dentro de exitfn1
```

Como verás, el kernel de UNIX no sabe nada de nada de tus *atexit*. Simplemente la función de librería *exit(3)* se ocupa de hacer los honores. ¿Qué pasaría si llamas a *exit* desde una función instalada con *atexit*? ¡Pruébalo! ¿Entiendes por qué?

7. Errores

En el programa que hemos hecho y en los siguientes vamos a hacer muchas llamadas al sistema. Bueno, en realidad muchas serán llamadas a la librería de C y otras serán llamadas al sistema. Nos da un poco igual (salvo por la sección del manual en que están documentadas, que será la 2 para llamadas al sistema y la 3 para llamadas a la librería de C). En cualquier caso, son llamadas para utilizar UNIX.

En muchos casos no habrá problema en la llamada que hagamos podrá hacer su trabajo. Pero en otros casos no será así. O bien habremos utilizado argumentos incorrectos en la llamada, o bien tendremos un problema de permisos o de otro tipo. Por ejemplo, cada proceso tiene un directorio de trabajo (como sabes) y podemos utilizar la llamada *chdir(2)* para cambiarlo. ¡Pero es posible que intentemos cambiar a un directorio que no existe!

Todas las llamadas devuelven un valor que, entre otras cosas, nos indicará al menos si han podido o no hacer su trabajo. Es responsabilidad nuestra comprobar dicho valor y actuar en consecuencia. Habitualmente el valor devuelto por una función suele ser un valor absurdo en el caso de errores. Si la función devuelve un puntero, devolverá `NULL`, si devuelve un entero positivo, devolverá `-1`, etc. La tradición en UNIX es que si el valor devuelto es sólo para indicar un error, suele devolverse `-1` en caso de error y `0` en caso de éxito. Las páginas de manual de cada función indican qué se devuelve en caso de error.

Cuando programes en C deberías seguir el mismo convenio. Todas tus funciones deberían informar al que las llama de si hubo algún error o no (salvo en aquellos casos en que nunca pueda producirse un error y en aquellos casos en que si se produce un error la función termine la ejecución de todo el programa).

Debes **siempre comprobar si hubo errores** en tus programas, en todas las llamadas. Si no lo haces, el programa será un poltergueist. ¿Y qué debería hacerse cuando hubo un error? No hay una respuesta correcta a esta pregunta. Depende de lo que estés programando y de qué función tenga el error. Simplemente piensa qué te gustaría que pasara si en una llamada sufres un error. Imagina que el programa lo has tomado prestado y piensa en lo que crees que debería hacer en cada caso.

Una vez has comprobado que una llamada no ha podido hacer su trabajo (¡eso es lo que significa que sufrió un error!, los errores no son magia), deberías ver a qué se debió el error. En UNIX, puedes utilizar la global `errno` para ver el código del error (un entero) y puedes utilizar *strerror(3)* para obtener un string correspondiente a dicho código.

(c) 2016 Francisco J. Ballesteros

Por ejemplo, este programa intenta cambiar de directorio:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    if (chdir("foo") < 0){
        printf("errno is %d\n", errno);
        printf("err string is '%s'\n", strerror(errno));
        exit(1);
    }
    /* ... do other things ... */
    exit(0);
}
```

Suponiendo que el ejecutable es `err` y que no existe el directorio `foo`, cuando lo ejecutamos...

```
unix$ err
errno is 2
err string is 'No such file or directory'
unix$
```

En la práctica, deberíamos imprimir el nombre del programa, lo que intentábamos hacer cuando hemos sufrido el error, y cuál es la causa:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    if (chdir("foo") < 0){
        fprintf(stderr, "%s: chdir: foo: %s\n", argv[0], strerror(errno));
        exit(1);
    }
    /* ... do other things ... */
    exit(0);
}
```

Si ahora ejecutamos el programa y este no puede cambiar su directorio, obtenemos un mensaje que ayudará a resolver el problema:

```
unix$ err
err: chdir: foo: No such file or directory
unix$
```

(c) 2016 Francisco J. Ballesteros

Esto es tan habitual, que la función *warn(3)* hace justo eso. Por ejemplo:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    if (chdir("foo") < 0){
        warn("chdir: %s", "foo");
        exit(1);
    }
    /* ... do other things ... */
    exit(0);
}
```

Y ahora tenemos...

```
unix$ err
err: chdir: foo: No such file or directory
unix$
```

Como verás, `warn` se usa de un modo similar a *printf(3)* e imprime un mensaje de error indicando el nombre del programa y el string de error correspondiente a `errno`. Si has leído la página de manual *warn(3)* (¡Cosa que deberías haber hecho en este punto!) habrás visto que la función *err(3)* es similar a `warn` pero llama a `exit` tras imprimir el mensaje con el estatus que le pasas en el primer argumento.

Un último comentario. No tiene sentido utilizar `errno` si no es justo después de una llamada que ha fallado (y ha indicado su error con el valor de retorno). Sólo cuando una llamada a UNIX tiene un problema, dicha llamada actualiza `errno` para informar de la causa del error.

8. Variables de entorno

Otra forma de darle información a un proceso (además de usando sus argumentos) es utilizar las llamadas **variables de entorno**. Cada proceso tiene un array de strings de tal forma que cada string tiene el aspecto

"*nombre*=*valor*"

y define una variable de entorno con nombre "*nombre*" y valor "*valor*".

Cuando UNIX inicializa la memoria para un nuevo programa se ocupa de que dicho array esté inicializado con las variables de entorno que se han indicado en la llamada al sistema utilizada para ejecutar un nuevo programa. Además de `argv`, ahora tienes que considerar que tienes un array de variables de entorno.

El propósito de estas variables es definir elementos que se desea que estén disponibles para los programas que ejecutas sin tener que pasar dichos elementos como argumento todas las veces. Naturalmente, tanto el nombre de las variables como su valor son strings, dado que esta abstracción (variables de entorno) consiste en el array de strings que hemos descrito.

Podemos definir variables de entorno utilizando el shell. Por ejemplo:

```
unix$ v=33
```

Cuando el shell ve que una palabra en una línea de comandos comienza por un "`$`", entiende que se desea

(c) 2016 Francisco J. Ballesteros

utilizar el valor de la variable de entorno cuyo nombre sigue y cambia dicha palabra por el valor de la variable (¡que siempre es un string!).

Por ejemplo, ya conoces *echo(1)* y sabes que simplemente escribe sus argumentos tal cual los recibe en `argv`, sin ningún tipo de magia. Pues bien, mira lo que escribe *echo*:

```
unix$ v=33
unix$ echo $v
33
unix$
```

Te resultará útil definir variables para nombres que utilices a menudo. Por ejemplo, si sueles cambiar de directorio a un directorio dado, puedes utilizar una variable de entorno para escribir menos:

```
unix$ d=/un/path/muy/largo
unix$ cd $d
unix$
```

Es tan habitual utilizar el directorio *home* de un usuario, que cuando haces un *login* se define una variable de entorno `HOME` que contiene el path de tu directorio *home*. Así pues:

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

El shell expande variables de entorno (reemplaza `$x` por el valor de la variable `x`) en cualquier sitio de la línea de comandos. Fíjate en esto:

```
unix$ p=pwd
unix$ $p
/home/nemo
unix$
```

Pero, si intentamos utilizar este truco para ejecutar "`ls -l`" nos llevamos una sorpresa...

```
unix$ l=ls -l
sh: -l: command not found
```

Necesitamos escribir *una sóla palabra* tras el operador "`=`". Esta es la sintaxis del shell para definir variables. Como hay un espacio entre "`ls`" y "`-l`", el shell ha intentado ejecutar "`-l`" como uno comando.

¿Y si probamos...?

```
unix$ l=ls-l
unix$
```

¡Ha funcionado!, ¿O no?

```
unix$ $l
sh: ls-l: command not found
unix$
```

¿Comprendes lo que ha sucedido? Seguro que sí, y, si no es así, piensa... ¿Cuál es el nombre del comando

(c) 2016 Francisco J. Ballesteros

que has ejecutado?

La solución a nuestro problema es utilizar sintaxis de shell para indicarle que cierto texto que escribimos en la línea de comandos ha de entenderse como una sóla palabra. Veámoslo:

```
unix$ l='ls -l'
unix$
```

Y ahora podemos...

```
unix$ $l
total 240
drwxr-xr-x   2 nemo  staff   1156 Jul 25 12:36 zxbib
drwxr-xr-x   5 nemo  staff   1734 Jul 25 13:01 zxdoc
...
```

Vamos a utilizar nuestro programa `eco2` para ver qué argumentos pasa el shell cuando utilizamos las comillas.

```
unix$ eco2 -v 'ls -l' ls -l
[ls -l] [ls] [-l]
```

Lo escrito entre comillas simples está en un sólo string en el vector `argv` de `eco2`. Esto es, el shell ha tomado literalmente lo que hay entre comillas simples como una sóla palabra. Ya lo sabías si recuerdas el primer tema de este curso.

También podemos utilizar comillas dobles para hacer que el shell tome su contenido como una sóla palabra. La diferencia entre estas y las simples radica en que el shell, en el caso de las comillas dobles, expande variables de entorno dentro de ellas. Por ejemplo:

```
unix$ cmd=ls
unix$ arg=-l
unix$ line="$cmd $arg"
unix$ $line
total 240
drwxr-xr-x   2 nemo  staff   1156 Jul 25 12:36 zxbib
drwxr-xr-x   5 nemo  staff   1734 Jul 25 13:01 zxdoc
...
```

En cambio...

```
unix$ cmd=ls
unix$ arg=-l
unix$ line='$cmd $arg'
unix$ $line
sh: $cmd: command not found
unix$ echo $line
$cmd $arg
```

En un programa en C podemos consultar variables de entorno llamando a *getenv(3)*. Por ejemplo, este programa cambia su directorio actual al directorio casa e imprime el path de dicho directorio antes de continuar con su trabajo.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>

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

    home = getenv("HOME");
    if (home == NULL){
        fprintf(stderr, "we are homeless\n");
        exit(1);
    }
    if (chdir(home) < 0) {
        err(1, "can't cd to home");
    }
    printf("working at home: %s\n", home);

    // ... do some other things...

    exit(0);
}
```

Si lo tenemos compilado en `env` y lo ejecutamos, podemos ver lo que hace...

```
unix$ env
working at home: /home/nemo
unix$
```

Si la variable no está definida, `getenv` devuelve `NULL`. Esto no es un error, y `errno` no se actualizará en tal caso. Simplemente la variable puede no estar definida. O dicho de otro modo... puede que ningún string en tu vector de variables de entorno comience por "`HOME=`" (aunque no es lo que uno espera en UNIX para la variable `HOME`).

¿Qué haría el programa si lo cambiamos para que ejecute

```
home = getenv("$HOME");
```

en lugar de hacer que hacía antes? ¡Pruébalo! ¿Puedes ver por qué? ¿Cómo se llama la variable? Recuerda que "`$HOME`" es sintaxis del shell. ¿Ejecuta el shell en algún caso cuando tu programa en C ejecuta la línea anterior? ¡Desde luego que no!

¿Y qué haría si lo cambiamos para que utilice

```
if (chdir("$HOME") < 0) {
    err(1, "can't cd to home");
}
```

o para que utilice

```
if (chdir("HOME") < 0) {
    err(1, "can't cd to home");
}
```

en lugar de lo que teníamos antes?

Para definir una variable de entorno en el proceso en que ejecutamos un programa en C podemos utilizar la

(c) 2016 Francisco J. Ballesteros

función *putenv(3)* o *setenv(3)*. Ambas están descritas en *getenv(3)*, así que si has seguido con la buena costumbre de leer la página de manual de cada llamada la primera vez que la usas, ya las conoces. Esto es un ejemplo en cualquier caso:

```
#include <stdio.h>
#include <stdlib.h>

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

    putenv("tempdir=/tmp");
    temp = getenv("tempdir");
    if (temp == NULL){
        fprintf(stderr, "no temp dir\n");
        exit(1);
    }
    printf("temp dir is: %s\n", temp);

    exit(0);
}
```

Y esta es su ejecución:

```
unix$ penv
temp dir is: /tmp
```

Naturalmente, sería absurdo utilizar la variable de entorno `tempdir` como lo hemos hecho si no pensamos ejecutar nuevos procesos desde nuestro programa en C. Si tan sólo queremos definir una variable para guardar el path de un directorio temporal, C ya tiene variables.

Hasta ahora todo bien. Pero... ¿Puedes explicar esto?

```
unix$ penv
temp dir is: /tmp
unix$ echo $tempdir
unix$
```

Resulta que tras ejecutar nuestro programa en C que define una variable de entorno, ¡el shell no la tiene definida! Pero esto es normal. Piensa que la variable la define el proceso que ejecuta el programa en C, y que el shell es otro proceso. Cada proceso tiene sus propias variables de entorno. ¿Lo ves normal ahora?

Cuando no entiendas algo, piensa que no hay magia y piensa en qué programas intervienen en lo que estás haciendo y qué hace cada uno paso a paso. Si sigues sin poder explicar el resultado es que te estás perdiendo algo respecto a lo que hace cada programa. Recurre al manual y haz programas de prueba que intenten explicar tus hipótesis respecto a qué está pasando.

En relación con esto, resulta interesante mirar lo que sucede en este otro caso:

(c) 2016 Francisco J. Ballesteros

```
unix$ x=foo
unix$ echo $x
foo
unix$ sh
unix$ echo $x

unix$ exit
unix$ echo $x
foo
unix$
```

Hemos ejecutado un shell (ejecutando el comando `sh`) y dicho shell no tiene definida la variable `x` (`$x` no tiene nada). Cuando terminamos dicho shell con `exit` volvemos al shell original y ahí si que tenemos la variable de entorno definida.

Es como si las variables que defines fuesen locales al shell y los comandos que ejecutamos desde ese shell ya no las tienen. Vamos a comprobarlo con un programa en C:

```
#include <stdio.h>
#include <stdlib.h>

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

    x = getenv("x");
    if (x == NULL){
        fprintf(stderr, "no x\n");
        exit(1);
    }
    printf("%s\n", x);

    exit(0);
}
```

Si lo compilamos y dejamos el ejecutable en `penvx` podemos ver lo que sucede:

```
unix$ x=foo
unix$ penvx
no x
unix$
```

Efectivamente, el shell no ha pedido a UNIX que ejecute `penvx` con la variable `x` definida en el entorno. Pero hay formas de hacer que lo haga:

```
unix$ export x
unix$ penvx
foo
unix$
```

El comando `export` es un comando construido dentro del shell. Es una *primitiva* o *builtin* del shell. Su propósito es indicarle al shell que *exporte* una o más variables de entorno a los procesos que ejecute dicho shell para ejecutar nuevos comandos.

Hoy día suele ser habitual hacer ambas cosas a la vez:

```
unix$ export x=foo
unix$ penvx
foo
unix$
```

9. Procesos y nombres

Ya sabes que un proceso es tan sólo una abstracción que implementa el kernel del sistema operativo. Dentro del kernel tenemos un array de records de tal forma que cada record será de tipo `Proc` (o cualquier otro nombre) y definirá el tipo de datos *proceso*.

Como te imaginarás en este punto, cada uno de esos records contiene el path para el directorio actual en que ejecuta el proceso y cualquier otra cosa que UNIX necesite recordar sobre ese proceso. Por ejemplo, qué segmentos de memoria utiliza. Y, naturalmente, los segmentos son también una abstracción y serán tan sólo records que contienen la información que UNIX necesite saber sobre cada uno de ellos. Así de simple.

El nombre de un proceso es parte de la abstracción *proceso* en UNIX. Pero dicho nombre *no* es el nombre del programa que está ejecutando (lo que sería `argv[0]` en la función `main` de dicho programa). El nombre de un proceso es un número entero que identifica el proceso y se conoce como **identificador de proceso** o *process id* o *pid*.

Cuando UNIX crea un proceso le asigna un *pid* que ningún otro proceso ha utilizado antes. Es como un "DNI" para el proceso. Todas las llamadas al sistema que necesitan que indiques sobre qué proceso deben operar reciben un *pid* para que nombres el proceso sobre el que han de trabajar. Pero recuerda que hay muchas llamadas que operan sobre el proceso que hace la llamada y, como es natural, no necesitan que indiques sobre qué proceso han de actuar.

Por ejemplo, cuando un programa llama a `chdir`, UNIX sabe qué proceso está haciendo la llamada (puesto que ha sido UNIX el que lo puso a ejecutar), y dicha llamada opera sobre el proceso en cuestión.

El siguiente programa utiliza la llamada al sistema *getpid(2)* para averiguar el *pid* del proceso que ejecuta el programa.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char* argv[])
{
    pid_t p;

    p = getpid();
    printf("pid is %d\n", p);
    exit(0);
}
```

El tipo `pid_t` es tan solo un tipo de `int`, no hay magia en eso tampoco. Vamos a ejecutar el programa varias veces, tras compilarlo en `pid`:

```
unix$ pid
pid is 91795
unix$ pid
pid is 91800
unix$ pid
pid is 91805
unix$
```

Y cada vez tendrá un nuevo *pid*. En un mismo programa, pero los procesos son distintos.

El comando *ps(1)* lista los procesos que están ejecutando

```
unix$ ps
  PID TT  STAT       TIME COMMAND
15891 p0  Ss      0:00.11 -ksh (ksh)
14610 p0  R+      0:00.00 ps
12349 C0- I       0:00.00 acme
unix$
```

La salida varía mucho de un tipo de UNIX a otro. En este caso el shell que ejecutamos no es `sh`, sino `ksh` y tenemos un editor en el programa `acme` que está ejecutando. Además, podemos ver cómo `ps` está ejecutando también.

Habitualmente `ps` lista sólo los procesos que ejecutan a nombre del mismo usuario que ejecuta `ps`. Pero `ps` tiene muchas opciones y te permite listar todos los procesos que ejecutan en la máquina así como ver muchas otras cosas sobre cada proceso (cuánta memoria virtual está utilizando cada proceso, cuánta memoria real, qué tiempo ha consumido de CPU, cuánto hace que está ejecutando, etc.). Lo mejor es que utilices el manual y recuerdes que en el caso de `ps` las opciones y las columnas que imprime para cada proceso suelen variar de un tipo de UNIX a otro.

Eso sí, lo normal suele ser que se imprima el *pid* de cada proceso (en la primera columna en la mayoría de los UNIX) y el vector de argumentos para cada proceso (normalmente al final de cada línea).

Por ejemplo, en el UNIX que estamos utilizando (un BSD) podemos utilizar estos flags en `ps` para listar todos los procesos y más información sobre cada uno de ellos:

```
unix$ ps -auxw
USER       PID %CPU %MEM   VSZ   RSS TT  STAT  STARTED       TIME COMMAND
elf      15891  0.0  0.0   680   856 p0  Ss    12:48PM    0:00.11 -ksh (ksh)
elf      27386  0.0  0.0   420   452 p0  R+    12:56PM    0:00.00 ps -auw
elf      12349  0.0  0.0   624   692 C0- I     14Jul16    0:00.00 ksh -c /zx/bin/xcmd -s sh -v
elf      26019  0.0  0.0 17720  5040 C0- I     14Jul16    2:36.19 /zx/bin/xcmd
root      5830  0.0  0.0   492  1148 C0  Is+   14Jul16    0:00.01 /usr/libexec/getty ttyC0
root     15869  0.0  0.0   500  1148 C1  Is+   14Jul16    0:00.00 /usr/libexec/getty ttyC1
...
```

Aquí, `VSZ` es la cantidad de memoria virtual y `RSS` es la cantidad de memoria física en uso. La columna `STAT` describe el estado de planificación del proceso (ya sabes... *ejecutando*, *listo para ejecutar*, *bloqueado*). Mira el manual de *ps* para ver qué significan las cosas en la salida que obtengas en tu UNIX.

10. Usuarios

¿A nombre de qué usuario ejecutamos? Veámoslo...

(c) 2016 Francisco J. Ballesteros

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

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

    uid = getuid();
    printf("uid is %d\n", uid);
    exit(0);
}
```

El nombre de usuario para UNIX es otro entero (como sabes) o *identificador de usuario*, o *uid*. La llamada *getuid(2)* te permite obtener el *uid* del usuario a nombre de quien ejecuta el proceso. Si compilamos el programa en `guid`, podemos ejecutarlo...

```
unix$ guid
uid is 501
unix$
```

Si ejecutamos el comando *id(1)* podemos ver que es correcto:

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

En este caso y en este UNIX, mi usuario es el `501`. Igualmente, podemos averiguar a nombre de qué grupo de usuarios está ejecutanto nuestro proceso.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

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

    gid = getgid();
    printf("gid is %d\n", gid);
    exit(0);
}
```

Si compilamos el programa en `ggid`, podemos ejecutarlo...

```
unix$ ggid
gid is 20
unix$
```

¡Ya sabes! Tanto el *uid* como el *gid* son atributos de cada proceso.

¿Y qué nombre de usuario corresponde a cada *uid*? En realidad, estamos adentrándonos en un campo que varía de un UNIX a otro. En general, las cuentas de usuario se abren editando un fichero llamado `/etc/passwd`, y en dicho fichero se suele incluir una línea para cada usuario que detalla su nombre, *uid*, *gid* para cada grupo al que pertenece el usuario, directorio casa, shell que ejecuta el usuario cuando hace un

(c) 2016 Francisco J. Ballesteros

login, etc. Del mismo modo, el fichero `/etc/group` suele utilizarse para incluir una línea por cada grupo de usuarios con el nombre del grupo y el *gid* del grupo, al menos.

En la mayoría de los UNIX podemos utilizar *getpwuid* para recuperar desde C un record que describe una entrada en *passwd* para un *uid* dado.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>
#include <uuid/uuid.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int uid;
    struct passwd *p;

    uid = getuid();
    p = getpwuid(uid);
    if (p == NULL) {
        err(1, "no passwd for uid");
    }
    printf("user name is %s\n", p->pw_name);
    exit(0);
}
```

Lee *getpwuid(3)* y echa un vistazo a las otras funciones y a los campos de `passwd`. Pero antes vamos a ejecutar este programa:

```
unix$ uidname
user name is nemo
unix$
```

Cualquier usuario puede cambiar su password utilizando *passwd(1)*. Naturalmente el comando *passwd* ejecuta a nombre del usuario que lo ejecuta. La pregunta entonces es... ¿Cómo puede tener *passwd* permisos para cambiar el password? Si tu usuario tiene permisos para editar `/etc/passwd` (o el fichero donde quiera que se guarden los passwords, encriptados) entonces tu usuario tendría acceso a todas las cuentas de usuario del sistema, lo que no es razonable salvo para el superusuario (o *root*, o *uid* `0`).

La respuesta suele ser que el fichero con el ejecutable de *passwd* tiene un permiso puesto, el *set uid bit*, que indica que dicho comando debe ejecutar de forma efectiva a nombre del dueño del fichero, y no a nombre del usuario que ejecuta dicho fichero. Dado que `/bin/passwd` (o el fichero de que se trate) pertenece normalmente a *root*, cualquier usuario puede cambiar su password.

Fíjate en esto:

```
unix$ which passwd
/usr/bin/passwd
unix$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  bin  23800 Mar  8  2015 /usr/bin/passwd*
```

El comando *which(1)* imprime el nombre del fichero que corresponde a un comando dado. Como puedes

(c) 2016 Francisco J. Ballesteros

ver, hay una `s` como permiso de ejecución para el dueño de `passwd`. Eso quiere decir que cualquier usuario que ejecute dicho fichero obtiene un proceso a nombre de *root*.

Podemos poner o quitar ese permiso como cualquier otro, siempre que tengamos permiso para ello:

```
unix$ cc -o eco eco.c
unix$ chmod u+s eco
```

Existe otro bit similar para el grupo, se llama el *set gid bit*, y se puede activar como en...

```
unix$ chmod g+s eco
```

Ahora podemos decir que en realidad los procesos tienen dos *uid* y dos *gid*. Tienen los reales y tienen los *efectivos*. Las llamadas *setuid(2)* y *seteuid(2)* permiten cambiar los *uid* real y efectivo de un proceso, y las llamadas *setgid(2)* y *setegid(2)* permiten cambiar los *gid* real y efectivo.

Como podrás suponer, en realidad son los identificadores efectivos los que se utilizan para comprobar permisos. Los reales suelen utilizarse para saber quién está en realidad ejecutando qué.

El comando *su(1)* (o *switch user*) permite ejecutar un shell a nombre de otro usuario (de *root* si no se indica nombre de usuario) y naturalmente utiliza el mecanismo de *set uid* para conseguirlo. Por ejemplo:

```
unix$ su
Password:
unix# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon)
unix# exit
unix$
```

11. ¿Qué más tiene un proceso?

En este punto sabes que los procesos tienen un *pid*, segmentos de memoria, valores para los registros (mientras no ejecutan, ¡cuando ejecutan dichos valores se guardan en los registros de verdad!), un directorio actual, variables de entorno, el identificador de usuario o *uid* a nombre del cual ejecutan, etc.

De aquí en adelante veremos otros recursos que tienen los procesos tales como ficheros abiertos y otros muchos. Piensa siempre que se trata de otros campos en el record que implementa cada proceso e intenta imaginar las estructuras de datos que los implementan. Es una buena forma de que vea que no hay magia por ningún lado.

Pero... ¿Cómo vas a ser capaz de recordar todo esto? ¡No lo hagas! ¡Usa el manual!

12. Cuando las cosas se tuercen...

De vez en cuando un proceso ejecutará un programa que tendrá un error y no hará lo que esperamos o hará algo demasiado grave que casuse que UNIX lo mate (por ejemplo, accediendo a una dirección de memoria a la que no tiene permiso para acceder por estar fuera de un segmento o por ser una escritura de un segmento de sólo lectura).

En la mayoría de los errores, incluir algún `printf` para depurar tras pensar en qué puede estar fallando es la mejor herramienta. En otros casos, necesitaremos ayuda para ver qué sucede. Para eso podemos utilizar un depurador. Cuando un programa hace algo que cause que UNIX lo mate, UNIX permite que se haga un volcado del estado de la memoria del proceso a un fichero (y del valor de los registros en el momento de la muerte). Dicho fichero suele llamarse `core` dado que es un volcado de la memoria y, cuando se hizo

(c) 2016 Francisco J. Ballesteros

UNIX, la memoria podía ser de núcleos (o "cores") de ferrita!). Según una de las personas que hizo UNIX, ¡las cosas empezaron a ir mal cuando los bits dejaron de verse a simple vista!

Vamos a hacer un programa que se comporte mal.

```
#include <stdlib.h>

static void
clearstr(char *p)
{
    p[0] = 0;
}

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

    p = NULL;

    clearstr(p);
    // ... do other things here...
    exit(0);
}
```

Y vamos a ejecutarlo:

```
unix$ bad
Segmentation fault: 11
unix$
```

El programa ha intentado utilizar una dirección de memoria que no tiene y la causado una violación de segmento. Cuando pasa esto, el hardware eleva una excepción al intentar traducir la dirección de memoria y el manejador de la excepción (el kernel) ve que el proceso no debería hacer eso. Como resultado, el kernel termina la ejecución del proceso. En nuestro caso, el shell se ha dado cuenta de lo que ha sucedido y ha impreso un mensaje para informar de ello.

Para conseguir un `core`, hemos de cambiar uno de los límites que tiene el proceso. Concretamente, el límite que indica el tamaño máximo de `core` que queremos obtener. Una vez más, los límites son también atributos del proceso. Desde C puedes usar *setrlimit(2)* para cambiar un límite y *getrlimit(2)* para consultar el valor actual. Desde el shell podemos ver los límites que tenemos con el comando *ulimit(1)*:

```
unix$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```

(c) 2016 Francisco J. Ballesteros

Y cambiar el tamaño máximo de core como sigue:

```
unix$ ulimit -c unlimited
unix$
```

Ahora podemos ejecutar nuestro programa roto una vez más:

```
unix$ bad
Segmentation fault: 11 (core dumped)
unix$
```

El lugar en que UNIX guarda los ficheros `core` varía de un UNIX a otro. Hace tiempo era el directorio actual del proceso que muere. Hoy día depende mucho del tipo de UNIX. Por ejemplo, en MacOS (OS X), el manual dice...

```
CORE(5)                   BSD File Formats Manual                   CORE(5)

NAME
     core -- memory image file format

SYNOPSIS
     #include <sys/param.h>

DESCRIPTION
     A small number of signals which cause abnormal termination of a process
     also cause a record of the process's in-core state to be written to disk
     for later examination by one of the available debuggers.  (See
     sigaction(2).)  This memory image is written to a file named by default
     core.pid, where pid is the process ID of the process, in the /cores
     directory, provided the terminated process had write permission in the
     directory, and the directory existed.
```

Así pues, en este sistema:

```
unix$ ls -l /cores
total 1234232
-r--------  1 nemo  admin  631926784 Aug 19 15:25 core.92367
```

Donde `92367` era el *pid* del proceso que murió.

Podemos utilizar un depurador para ver un volcado de la pila en el momento de la muerte. Esto suele ser mas que suficiente para avergüar la causa del problema:

(c) 2016 Francisco J. Ballesteros

```
unix$ lldb -c /cores/core.92367
(lldb) target create --core "/cores/core.92433"
Core file '/cores/core.92433' (x86_64) was loaded.
(lldb) bt
* thread #1: tid = 0x0000, 0x0000000105a53f6c
    bad'clearstr(p=0x0000000000000000) + 12
    at bad.c:10, stop reason = signal SIGSTOP
  * frame #0: 0x0000000105a53f6c
    bad'clearstr(p=0x0000000000000000) + 12
    at bad.c:10
    frame #1: 0x0000000105a53f57
    bad'main(argc=1, argv=0x00007fff5a1acb20) + 39
    at bad.c:20
    frame #2: 0x00007fff94ad65ad libdyld.dylib'start + 1
(lldb) q
unix$
```

Hemos utilizado el depurador *lldb(1)*, que tiene la opción `-c` para indicarle que queremos inspeccionar un *core dump* (un fichero `core`). *LLdb* es un shell en el que podemos escribir comandos para depurar. El comando *bt* imprime un volcado o *backtrace* de la pila. El comando *q* termina la ejecución del depurador.

Como podrás ver, el program ha muerto en la función `clearstr` del ejecutable `bad`. Concretamente en la línea 10. A dicha función la ha llamado `main` en el ejecutable `bad`, desde la línea 20 de `bad.c`. Y a `main` lo ha llamado una función llamada `start` dentro del cargador de librerías dinámicas (mira el capítulo de introducción si no recuerdas lo que es una librería dinámica).

Pues bien, la línea 10 de `bad.c` es:

```
p[0] = 0;
```

Y podemos ver que el argumento de `clearstr`, `p`, tiene como valor `0`. Luego sucede que `p` es `NULL` y no podemos atravesar dicho puntero. Ese es el problema. Ahora habría que pensar en por qué ha sucedido y en qué deberíamos hacer para arreglarlo.

Un detalle importante es que para que el depurador pueda saber qué ficheros fuente y líneas corresponden a cada contador de programa, hay que pedir al compilador que incluya información de depuración en el ejecutable (que incluya una tabla con la correspondencia de nombre de fichero fuente y línea a contador de programa, y tal vez otra información como nombres de función y variables). Habitualmente, el flag `-g` del compilador de C consigue dicho efecto. Esto es, hemos compilado como sigue:

```
unix$ cc -g -o bad bad.c
```

Si no incluimos la información de depuración, el depurador no puede hacer magia. Podemos utilizar el comando *strip(1)* que elimina la información de depuración de un ejecutable y ver el resultado:

```
unix$ strip bad
unix$ bad
Segmentation fault: 11 (core dumped)
unix$
```

Y ahora...

(c) 2016 Francisco J. Ballesteros

```
unix$ ls /cores
core.92473
unix$ lldb -c /cores/core.92473
(lldb) target create --core "/cores/core.92473"
Core file '/cores/core.92473' (x86_64) was loaded.
(lldb) bt
* thread #1: tid = 0x0000, 0x00000001014c4f6c
    bad‘clearstr + 12, stop reason = signal SIGSTOP
  * frame #0: 0x00000001014c4f6c
    bad‘clearstr + 12
    frame #1: 0x00000001014c4f57
    bad‘main + 39
    frame #2: 0x00007fff94ad65ad libdyld.dylib‘start + 1
```

Como verás, sólo podemos ver el contador de programa en cada llamada registrada en la pila (en cada *registro de activación* de la pila). Concretamente, el programa murió en `bad‘clearstr + 12`. Esto es, 12 posiciones más allá del comienzo de `clearstr` en el fichero ejecutable `bad`. Pero claro, no sabemos ni el fichero fuente ni la línea.

Lo más aconsejable es compilar siempre con `-g` y dejar la información de depuración en los ejecutables.

Otro depurador muy popular es `gdb`. Lo mejor es que utilizes el manual o *google* para averigüar cómo obtener un volcado de pila de un fichero `core` con tu depurador, y que localices el directorio en que tu UNIX deja los *core dumps*.

13. /proc

Existe otro interfaz para manipular procesos más allá de las llamadas al sistema habituales para ello. Concretamente, hay un directorio en (la mayoría de los sistemas) UNIX que aparenta tener ficheros relacionados con procesos:

(c) 2016 Francisco J. Ballesteros

```
unix$ ls /proc
1       153     25    43    701         dma          mtrr
10      154     26    44    778         driver       net
1001    16      27    45    8           execdomains  pagetypeinfo
1002    17      28    46    89          fb           partitions
103     174     29    47    898         filesystems  sched_debug
1046    175     3     48    9           fs           schedstat
1071    18      30    5     90          interrupts   scsi
11      19      31    50    900         iomem        self
1130    2       32    516   903         ioports      slabinfo
12      20      325   52    908         irq          softirqs
13      21      33    522   909         kallsyms     stat
14      21102   330   53    911         kcore        swaps
143     21104   34    54    930         key-users    sys
144     21227   35    541   acpi        keys         sysrq-trigger
145     21246   36    55    asound      kmsg         sysvipc
146     21247   368   5627  buddyinfo   kpagecount   timer_list
147     21248   369   613   bus         kpageflags   timer_stats
148     21301   37    629   cgroups     loadavg      tty
149     21302   38    637   cmdline     locks        uptime
15      21317   39    67    consoles    mdstat       version
150     215     40    68    cpuinfo     meminfo      version_signature
151     22      41    69    crypto      misc         vmallocinfo
15160   23      42    7     devices     modules      vmstat
152     24      421   70    diskstats   mounts       zoneinfo
```

En este caso, hemos utilizado un sistema Linux. Los ficheros en `/proc` no son ficheros reales en el disco. UNIX se los inventa para reflejar el estado de los procesos que están ejecutando y para dejarte averigüar cosas sobre ellos e incluso operar sobre ellos. Aún más, no sólo para operar sobre procesos, sino para operar sobre el sistema entero.

Cuando un proceso lee o escribe uno de estos ficheros, UNIX hace que la operación sobre el fichero se comporte normalmente, pero UNIX inventa el resultado de la operación para hacer creer que dichos ficheros corresponden en cada momento al estado del sistema. Son ficheros *sintéticos*. Dicho de otro modo, son falsos y UNIX se los inventa en cada momento.

Como puedes ver por el listado de ficheros en `/proc` hay un directorio por proceso, siendo el nombre del directorio el pid del proceso. Dicho directorio contiene ficheros que permiten ver y cambiar datos del proceso en cuestión. Por ejemplo, en un sistema Linux:

(c) 2016 Francisco J. Ballesteros

```
unix$ ps
  PID TTY          TIME CMD
21422 pts/0    00:00:00 bash
21438 pts/0    00:00:00 ps
unix$ ls -l /proc/21422
unix$ ls  /proc/21422
attr        coredump_filter  gid_map    mountinfo   oom_score      schedstat  status
autogroup   cpuset           io         mounts      oom_score_adj  sessionid  syscall
auxv        cwd              limits     mountstats  pagemap        setgroups  task
cgroup      environ          loginuid   net         personality    smaps      timers
clear_refs  exe              map_files  ns          projid_map     stack      uid_map
cmdline     fd               maps       numa_maps   root           stat       wchan
comm        fdinfo           mem        oom_adj     sched          statm
unix$
```

Hemos listado el directorio que corresponde al proceso del shell que estábamos ejecutando. Podemos ver cual es la línea de comandos para dicho proceso:

```
unix$ cat /proc/21422/cmdline
-bashunix$
```

Dado que era un shell de login (ejecutado al hacer el login en el sistema), el programa *login* que lo creó utilizó `-bash` como valor para el primer argumento (`argv[0]` en `main` en dicho proceso). El convenio en UNIX es que si en un shell, `argv[0]` comienza por "`-`", entonces se trata de un shell de login (y habitualmente dicho shell leerá `$HOME/.profile` u otro fichero para ejecutar los comandos que contenga como parte del proceso de inicialización).

En `/proc/21422/maps` tenemos una descripción de los segmentos de memoria que utiliza dicho proceso:

```
unix$ cat /proc/21422/maps
00400000-004ef000 r-xp 00000000 08:01 17039362                           /bin/bash
006ef000-006f0000 r--p 000ef000 08:01 17039362                           /bin/bash
006f0000-006f9000 rw-p 000f0000 08:01 17039362                           /bin/bash
006f9000-006ff000 rw-p 00000000 00:00 0
01f99000-021a0000 rw-p 00000000 00:00 0                                  [heap]
...
7f3164b43000-7f3164b44000 rw-p 00000000 00:00 0
7fff170f1000-7fff17112000 rw-p 00000000 00:00 0                          [stack]
unix$
```

Puedes ver las direcciones de comienzo y fin para los segmentos de texto, datos inicializados de sólo lectura, datos inicializados, BSS, y pila (entre otros). Hemos borrado del listado los segmentos correspondientes a las librerías dinámicas que utiliza dicho proceso. Resulta interesante ver que los segmentos de texto y datos inicializados proceden de `/bin/bash` para este proceso. Como verás, el texto tiene permiso de ejecución y los datos tienen permiso de lectura al menos. Los datos que no son constantes están en un segmento con permisos de lectura escritura.

A la vista de esto, aunque en C, `"hola"` es un array de `char`, no deberías intentar cambiar su contenido. Es muy posible que dicho array esté guardado en memoria en un segmento de datos de sólo lectura durante la ejecución del programa.

El fichero `/proc/21422/status` tiene información interesante sobre el estado del proceso:

(c) 2016 Francisco J. Ballesteros

```
unix$ cat /proc/21422/status
Name:    bash
State:    S (sleeping)
Pid:    21422
PPid:    21421
...
voluntary_ctxt_switches:    398
nonvoluntary_ctxt_switches:    247
unix$
```

Entre otras cosas, puedes ver que actualmente el proceso está bloqueado (durmiendo) y que unas 398 veces ha hecho llamadas al sistema que han provocado que se le expulse del procesador. Además, unas 247 veces ha sido UNIX el que lo ha expulsado sin que el proceso hiciera ninguna llamada que causara la expulsión. Simplemente, llegaría una interrupción de reloj y UNIX decidiría que ya era hora de empezar a ejecutar un rato otro proceso.

Recuerda que `/proc` (en la mayoría de los UNIX) es una abstracción y no corresponde a datos reales guardados en ningún disco. UNIX se inventa esos ficheros respondiendo a las llamadas al sistema que operan sobre dichos ficheros para que aparentemente `/proc` contenga una representación del estado de los procesos y el sistema.

(c) 2016 Francisco J. Ballesteros

Capítulo 3: Ficheros

(c) 2016 Francisco J. Ballesteros

1. Entrada/Salida

Es importante saber cómo utilizar ficheros. En UNIX, es aún más importante dado que gran parte de los recursos, dispositivos y abstracciones del sistema aparentan ser ficheros. Por ejemplo, el teclado, la salida de texto en la pantalla, las conexiones de red, una impresora USB..., casi todo es un fichero o se comporta como tal.

Antes de UNIX la mayoría de los sistemas utilizaban abstracciones diferentes para cada dispositivo (para el teclado, la impresora, etc.). Una de las razones por las que UNIX se hizo popular es que utilizó la abstracción *fichero* como interfaz para todos ellos. Eso permite utilizar los comandos y llamadas al sistema que operan en ficheros para operar en casi cualquier dispositivo. ¡Al contrario que en MS-DOS y en Windows aunque estos sistemas sean posteriores!

En cuanto a ficheros contenidos en un disco, normalmente el disco se suele dividir en trozos más pequeños llamados particiones. Cada partición no es muy diferente a un disco: es una secuencia de bloques siendo cada bloque un array de bytes (normalmente de 512 bytes, 1KiB, 8KiB o 16KiB). UNIX guarda en cada partición estructuras de datos para implementar los ficheros. Se suele llamar **sistema de ficheros** al programa que implementa esa abstracción, el *fichero*, que suele ser parte del kernel. Cuando das formato a una partición para guardar ficheros, determinas el tipo de sistema de ficheros que creas. Veremos más sobre sistemas de ficheros más adelante.

En realidad ya conoces mucho sobre ficheros. En cursos previos has utilizado la librería de tu lenguaje de programación para abrir, leer y escribir ficheros. Y probablemente encontrases problemas que te costaba explicar. Ahora vamos a utilizar el interfaz suministrado por UNIX para manipular ficheros, que es casi el interfaz universal en todos los sistemas operativos, y verás cómo desaparecen los problemas tanto en UNIX como al usar cualquier otro lenguaje.

Considera `printf`. Es una función de librería documentada en *printf(3)* que imprime mensajes con formato. Escribe siempre en un fichero ¡incluso cuando escribes en la pantalla! y para ello llama a *write(2)*. Veamos un programa...

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    char    msg[] = "hello\n";
    int    n, nw;

    n = strlen(msg);
    nw = write(1, msg, n);
    if (nw != n) {
        err(1, "write");
    }
    exit(0);
}
```

Es algo más elaborado de lo que podría ser. El mensaje `msg` lo guardamos en un array de caracteres en lugar de utilizar `"hello\n"` directamente para que puedas ver lo que hace `write` lo más claramente posible. Vamos a ejecutarlo:

(c) 2016 Francisco J. Ballesteros

```
unix$ write
hello
unix$
```

¡Hace lo mismo que si utilizamos `printf("hello\n")` o `puts("hello")`!

La llamada *write(2)* escribe bytes en un fichero. El primer parámetro es un `int` que representa un fichero abierto en que se desea escribir. El segundo parámetro es una dirección de memoria, un puntero. Indica dónde tienes los bytes que deseas escribir en el fichero. El último parámetro es el número de bytes que deseas escribir. Tal y como indica el manual, el valor devuelto es el número de bytes que se han escrito y se considera un error que write escriba un número de bytes distinto al que se desea escribir. ¿Entiendes mejor el código ahora?

El fichero en que hemos escrito es simplemente `"1"` en este caso. Los ficheros tienen nombres, como sabes. Los nombres de fichero son strings que UNIX interpreta como paths absolutos o relativos, dependiendo de si comienzan por `"/"` o no. Pues bien, los ficheros abiertos que utiliza un proceso también tienen nombres. En este caso los llamamos **descriptores de fichero**.

Un *descriptor de fichero* es un entero que indica qué fichero abierto se desea usar. Cuando abres un fichero, UNIX te da un nuevo descriptor de fichero. Cuando lo cierras, dicho descriptor deja de existir. El descriptor de fichero es simplemente un índice en un array de ficheros abiertos. Este array lo mantiene UNIX para cada proceso. Mira la figura 10.

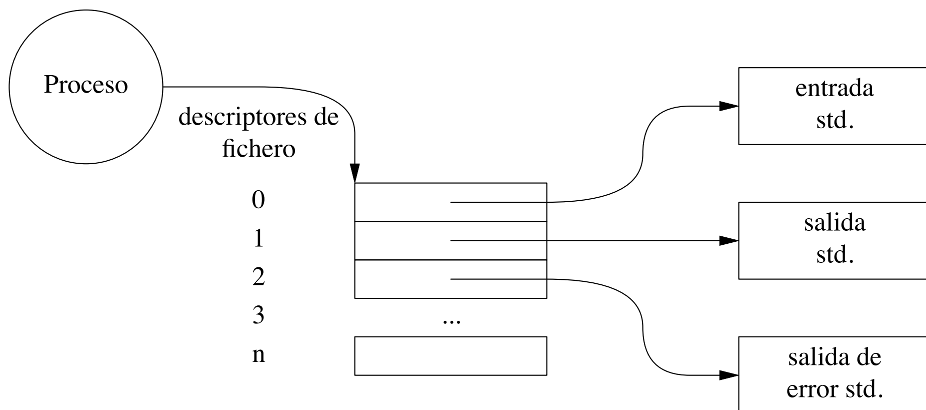

Figura 10: *Los descriptores de fichero son entradas en un array que apuntan a los ficheros abiertos de un proceso. Se utilizan para la entrada estándar, la salida estándar, la salida de error estándar y cualquier otro fichero abierto.*

El convenio en UNIX es que todos los procesos parten con tres descriptores de fichero abiertos: `0`, `1` y `2`. Corresponden a los ficheros llamados **entrada estándar**, **salida estándar** y **salida de error estándar**. La idea es que se espera que el programa lea sus datos de entrada del fichero abierto `0`, o de la entrada estándar o de *stdin*. Igualmente, se espera que el programa escriba cualquier resultado en el fichero abierto `1`, o en la salida estándar o en *stdout*. En cambio, los mensajes de error se escriben en la salida de error estándar, que corresponde con el descriptor `2`. La razón para tener una salida de error estándar separada de la salida estándar es evitar que se mezclen los resultados de la ejecución de un programa con los mensajes de error. Por ejemplo, podríamos ejecutar un programa guardando sus resultados en un fichero y, aún así, querremos que los mensajes de error aparezcan en la pantalla.

Cuando ejecutas un programa desde el shell en una ventana de tu sistema la entrada estándar es el teclado en dicha ventana y tanto la salida estándar como la salida de error estándar son la pantalla en dicha ventana. Esto es así a no ser que lo cambies. Recuerda cuando ejecutamos...

(c) 2016 Francisco J. Ballesteros

```
unix$ echo hola >unfichero
```

En este caso el shell habrá hecho que la salida estándar de `echo` sea `unfichero`. Ya veremos más adelante como funciona esto.

Para leer un fichero (abierto) se utiliza *read(2)*. Igual que `write`, esta función recibe tres argumentos: un descriptor de fichero, una dirección de memoria y un número entero. La llamada `read` lee bytes del fichero indicado por el descriptor de fichero y los deja en la memoria a partir de la posición indicada por el segundo argumento (un puntero). Como mucho se leen tantos bytes como indica el tercer argumento, pero podrían leerse menos. Veamos un programa que lee de la entrada estándar y escribe lo que lee en la salida estándar:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    char    buffer[1024];
    int    nr;

    nr = read(0, buffer, sizeof buffer);
    if (nr < 0) {
        err(1, "read");
    }
    if (write(1, buffer, nr) != nr) {
        err(1, "write");
    }
    exit(0);
}
```

Y así es cómo ejecuta...

```
unix$ read1
hola
hola
unix$
```

El primer "`hola`" lo hemos escrito nosotros. El segundo en cambio lo ha escrito el programa `read1`. Una vez lo ha escrito, el programa termina.

Como verás, el programa lee del descriptor `0` y escribe en el `1`. O dicho de otro modo, lee de la entrada y escribe en la salida lo que lee. Hasta que hemos escrito "`hola`" y pulsado *enter* el programa estaba esperando a que escribiéramos. O, más precisamente, el programa estaba dentro de la llamada a *read* y UNIX tenía el proceso bloqueado a la espera de tener algo que leer (esto es, no estaba ejecutando ni listo para ejecutar).

Cuando pulsamos teclas para escribir "`hola`" el teclado envía interrupciones que atiende UNIX. El manejador de dichas interrupciones ha ido guardando los caracteres correspondientes en un buffer hasta que hemos pulsado *enter*. En dicho momento, UNIX ha copiado los caracteres de dicho buffer hacia la memoria del proceso (para atender la llamada a *read* que estaba haciendo) y listo.

¡Esa es la razón por la que puedes borrar caracteres cuando escribes en el teclado! Cuando pulsas la tecla de borrar, UNIX lee el carácter (que es tan bueno como cualquier otro) y entiende que quieres eliminar el último carácter que había en el buffer de lectura de teclado. Cuando pulsas *enter*, UNIX entiende que dicha línea está lista para quien quiera que lea de teclado.

(c) 2016 Francisco J. Ballesteros

La mayoría de los programas en UNIX aceptan nombres de fichero como argumentos para trabajar con ellos y, lo normal, es que si no indicas ningún nombre de fichero el programa en cuestión trabaje con su entrada estándar.

Por ejemplo, en este caso

```
unix$ echo hola >/tmp/fich
unix$ cat /tmp/fich
hola
unix$ cat /tmp/fich /tmp/fich
hola
hola
unix$
```

el comando *cat* lee `/tmp/fich` y escribe su contenido en la salida estándar. Pero si llamamos a *cat* sin indicar ningún nombre de fichero...

```
unix$ cat
xxx
xxx
yyy
yyy
^D
unix$
```

cat se limita a leer de su entrada estándar hasta el fin de fichero y a escribir todo cuanto lea. Nosotros escribimos una línea con `xxx` en el teclado y *cat* la escribe en su salida. Después escribimos una línea con `yyy` en el teclado y *cat* la escribe en su salida. Hasta el fin de fichero.

¡Un momento! ¿Fin de fichero para el teclado? Pues si. El fichero que corresponde al teclado es una abstracción, ¿Recuerdas?. Cuando mantienes pulsado *control* y pulsas "`d`", (esto es, *control-d*, a veces escrito como `^D` como hemos hecho nosotros en el ejemplo) UNIX entiende que quieres que quien esté leyendo de teclado reciba una indicación de fin de fichero. Así pues, *cat* termina. (Acabamos de mentir respecto a lo que hace *control-d* pero desharemos la mentira en breve).

Cabe otra pregunta... ¿Cómo es que *cat* copia múltiples líneas y nuestro programa sólo una? La respuesta es simple si piensas que no hay magia y piensas en lo que hace *read(2)*. Cuando `read` lee los bytes que puede leer y te dice cuántos bytes ha leído, `read` ha terminado su trabajo. Nuestro programa llamaba a `read` una única vez, por lo que al leer de teclado ha leído una única línea.

Vamos a arreglarlo haciendo un nuevo programa.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    char    buffer[1024];
    int     nr;

    for(;;) {
        nr = read(0, buffer, sizeof buffer);
        if (nr < 0) {
            err(1, "read");
        }
        if (nr == 0) {
            break;     // EOF
        }
        if (write(1, buffer, nr) != nr) {
            err(1, "write");
        }
    }
    exit(0);
}
```

Si lo ejecutamos veremos que se comporta como *cat*:

```
unix$ readin
xxx
xxx
yyy
yyy
ˆD
unix$
```

Esta vez llamamos a `read` las veces que sea preciso para leer *toda* la entrada. En cada llamada pueden ocurrir tres cosas:

- Puede que leamos algunos bytes.
- Puede que no tengamos nada más que leer.
- Puede que suframos un error.

En el último caso la página de manual *read(2)* nos dice que `read` devuelve `-1` y actualiza `errno`. En dicho caso nuestro programa aborta tras imprimir un mensaje explicativo del error.

En el penúltimo caso (EOF) el programa termina con normalidad. Y, en el primer caso, el programa escribe en la salida los bytes que ha conseguido leer.

Recuerda que aunque tu desees leer *n* bytes, `read` no garantiza que pueda leerlos todos. Así pues, si deseas leer todo un fichero o un número dado de bytes, deberás llamar a `read` múltiples veces hasta que consigas leer todo lo que quieres. Un error frecuente cuando no se sabe utilizar UNIX (¡Cuando no se sabe leer!) es llamar a `read` una única vez.

(c) 2016 Francisco J. Ballesteros

2. Open, close, el terminal y la consola

Vamos a cambiar nuestro programa para que use *open(2)* y veamos que ocurre.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    char    buffer[1024];
    int    fd, nr;

    fd = open("/dev/tty", O_RDWR);
    if (fd < 0) {
        err(1, "open %s", "/dev/tty");
    }
    for(;;) {
        nr = read(fd, buffer, sizeof buffer);
        if (nr < 0) {
            close(fd);
            err(1, "read");
        }
        if (nr == 0) {
            break;    // EOF
        }
        if (write(fd, buffer, nr) != nr) {
            close(fd);
            err(1, "write");
        }
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Esto es lo que sucede al ejecutarlo:

```
unix$ readtty
hola
hola
caracola
caracola
^D
unix$
```

¡Lo mismo que al utilizar el descriptor `0` para leer y el descriptor `1` para escribir! Esta vez estamos utilizando como descriptor (para leer y escribir) el que nos ha devuelto `open`. Y hemos utilizado `open` para abrir el fichero `/dev/tty` para lectura/escritura. El primer parámetro de `open` es un nombre de fichero que queremos abrir y el segundo es un entero que has de interpretar como un conjunto de bits. El valor `O_RDWR` indica que queremos abrir el fichero para leer y escribir. El resultado de `open` es un entero que indica qué descriptor podemos utilizar para el nuevo fichero abierto. Por ejemplo, si en nuestro ejemplo

resulta que `open` ha devuelto `3`, tendríamos posiblemente unos descriptores como los que puedes ver en la figura 11.

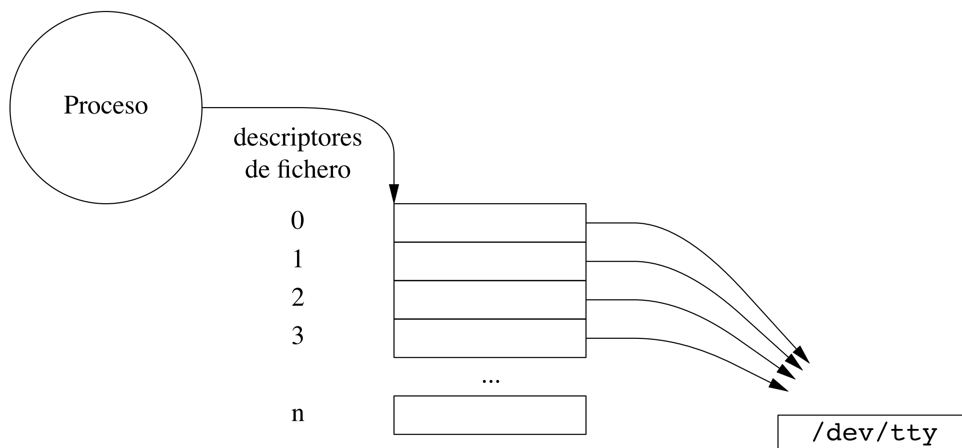

Figura 11: *Descriptores tras abrir el terminal usando* `open`.

Cuando abrimos un fichero, se espera que lo cerremos en el momento en que deje de sernos útil, cosa que se consigue llamando a *close(2)* con el descriptor de fichero que se desea cerrar. Y cuidado aquí... `close` podría fallar y hay que comprobar si ha podido hacer su trabajo o no. De no hacerlo, puede que no detectes que todas tus escrituras han alcanzado su destino. Una vez has cerrado un descriptor, podría ser que `open` en el futuro devuelva justo ese descriptor para otro fichero. Un descriptor de fichero es simplemente un índice en la tabla de ficheros abiertos del proceso.

En este punto, deberíamos preguntarnos... ¿Qué fichero es la entrada estándar? ¿Y la salida? La mayoría de las veces la entrada y la salida corresponden al fichero `/dev/tty`, que representa el *terminal* en que ejecuta nuestro programa. Es por esto que nuestro programa consigue el mismo efecto leyendo de `/dev/tty` que leyendo del descriptor `0` y escribiendo en `/dev/tty` en lugar de escribir en el descriptor `1`. Simplemente `0` y `1` ya se referían a `/dev/tty`.

Pero mira esto:

```
unix$ readin >/tmp/fich
hola
^D
unix$ cat /tmp/fich
hola
unix$
unix$ readtty >/tmp/fich
hola
hola
^D
unix$ cat /tmp/fich
unix$
```

Si utilizamos el programa que lee de la entrada estándar y escribe en la salida estándar (¡Que es lo que se espera de un programa en UNIX!) vemos que el programa se comporta de forma diferente a cuando utilizamos el programa que utiliza `/dev/tty` para leer y escribir en el. En este caso, hemos pedido al shell que ejecute `readin` enviando su salida estándar al fichero `/tmp/fich`, por lo que el programa que

(c) 2016 Francisco J. Ballesteros

escribe en `1` envía su salida correctamente a dicho fichero. En cambio, `readtty` sigue escribiendo en la ventana (en el terminal). ¡Normal!, considerando que dicho programa abre el terminal y escribe en el.

Cuando el sistema arranca, antes de que ejecute el sistema de ventanas, los programas utilizan la pantalla y el teclado. Ambos están abstraídos en el fichero `/dev/cons`, llamado así por ser la *consola* (Hace tiempo, las máquinas eran mucho mas grandes y tenían aspecto de mueble siendo la pantalla y el teclado algo con aspecto de consola).

Una vez ejecuta el sistema de ventanas (que es un programa como todo lo demás), éste se queda con la consola para poder leer y escribir y se inventa las *ventanas* como abstracción para que ejecuten nuevos programas. Igualmente, cuando un usuario remoto establece una conexión de red y se conecta para utilizar la máquina, se le asigna un *terminal* que es de nuevo una abstracción y tiene aspecto de ser un fichero similar a la consola.

En cualquier caso, `/dev/tty` es siempre el terminal que estamos utilizando. Si leemos, leemos del teclado. Cuando se trata de una ventana, el teclado naturalmente sólo escribe en esa ventana cuando la ventana tiene el *foco* (hemos dado click con el ratón en ella o algo similar).

Los ficheros que representan terminales pueden encontrarse en `/dev`:

```
unix$ ls /dev/tty*
/dev/ttyp0    /dev/ttyqa    /dev/ttys4    /dev/ttyte    /dev/ttyv8
/dev/ttyp1    /dev/ttyqb    /dev/ttys5    /dev/ttytf    /dev/ttyv9
/dev/ttyp2    /dev/ttyqc    /dev/ttys6    /dev/ttyu0    /dev/ttyva
/dev/ttyp3    /dev/ttyqd    /dev/ttys7    /dev/ttyu1    /dev/ttyvb
/dev/ttyp4    /dev/ttyqe    /dev/ttys8    /dev/ttyu2    /dev/ttyvc
/dev/ttyp5    /dev/ttyqf    /dev/ttys9    /dev/ttyu3    /dev/ttyvd
/dev/ttyp6    /dev/ttyr0    /dev/ttysa    /dev/ttyu4    /dev/ttyve
...
unix$
```

Pero para que sea trivial encontrar el fichero que corresponde al terminal que usa nuestro proceso, `/dev/tty` *siempre* corresponde al terminal que usamos. Eso sí, en cada proceso `/dev/tty` corresponderá a un fichero de terminal distinto. Esto no es un problema. Dado que UNIX sabe qué proceso está haciendo la llamada para abrir `/dev/tty`, UNIX puede dar "el cambiazo" perfectamente y hacer que se abra en realidad el terminal que está usando el proceso.

Ya ves que el **terminal de control** de un proceso (que es como se denomina) es en realidad otro de los atributos o elementos que tiene cada proceso en UNIX.

3. Ficheros abiertos

Resulta instructivo ver qué ficheros tiene abierto un proceso. Vamos a hacer un programa que abra un fichero y luego se limite a dormir durante un tiempo, para darnos tiempo a jugar con el.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

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

    fd = open("sleepfd.c", O_RDONLY);
    if (fd < 0) {
        err(1, "open: %s", "sleepfd.c");
    }
    sleep(3600);
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Primero vamos a ejecutarlo, pero pidiendo al shell que no espere a que termine antes de leer mas líneas de comandos...

```
unix$ sleepfd &
[1] 93552
unix$
```

El "`&`" al final de una línea de comandos es sintaxis de shell (de nuevo) y hace que shell continúe leyendo líneas de comandos sin esperar a que el comando termine. El shell ha sido tan amable de decirnos que el proceso tiene el pid `93552`, pero vamos a ver qué procesos tenemos en cualquier caso.

```
unix$ ps
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
93552 ttys002    0:00.00 sleepfd
...
unix$
```

Ahora que tenemos a `sleepfd` esperando, podemos utilizar el comando *lsof(1)* que lista ficheros abiertos. Este comando es útil tanto para ver qué procesos tienen determinado fichero abierto como para ver los ficheros abiertos de un proceso. Con la opción `-p` permite indicar el pid del proceso en que estamos interesados.

(c) 2016 Francisco J. Ballesteros

```
unix$ lsof -p 93552
COMMAND     PID USER   FD   TYPE DEVICE  SIZE/OFF    NODE NAME
sleepfd 93552 nemo  cwd    DIR    1,4      1020 7766070 /home/nemo/sot
sleepfd 93552 nemo  txt    REG    1,4      8700 7781483 /home/nemo/sot/sleepfd
sleepfd 93552 nemo  txt    REG    1,4    642448 4991292 /usr/lib/dyld
sleepfd 93552 nemo    0u   CHR   16,2  0t638511    1037 /dev/tty02
sleepfd 93552 nemo    1u   CHR   16,2  0t638511    1037 /dev/tty02
sleepfd 93552 nemo    2u   CHR   16,2  0t638511    1037 /dev/tty02
sleepfd 93552 nemo    3r   REG    1,4       398 7781474 /home/nemo/sot/sleepfd.c
unix$
```

La primera línea muestra que el proceso está usando `/home/nemo/sot`, que es un directorio (la columna `TYPE` muestra `DIR`). Mirando la columna llamada `FD`, vemos que indica `cwd`, lo que quiere decir que en realidad se trata del directorio actual (*current working directory*) del proceso.

La segunda línea muestra que también está usando `/home/nemo/sot/sleepfd`, ¡el ejecutable que hemos ejecutado!. Y la columna `TYPE` muestra `txt`, indicando que se está usando ese fichero como código (o texto) para paginar código hacia el segmento de texto, posiblemente. Igualmente, la tercera línea muestra que se está utilizando el código del enlazador dinámico `dyld` para suministrar código.

Las últimas cuatro filas son nuestro objetivo. Como puedes ver, la columna `FD` indica `0u`, `1u`, `2u` y `3r`. Además, puedes ver que `0`, `1` y `2` se refieren a `/dev/ttys002` (¡un terminal!). Estos tres son la entrada, salida y salida de error estándar de nuestro proceso. La cuarta fila indica que el descriptor `3` se refiere al fichero `sleepfd.c`, que es el fichero que nuestro programa ha abierto.

Para no dejar programas danzando inútilmente, vamos a matar nuestro proceso...

```
unix$ kill 93552
[1]+  Terminated: 15          sleepfd
unix$
```

El comando *kill(1)* puede utilizarse para matar procesos, basta darle el pid de los mismos. El shell, de nuevo, ha sido tan amable de informar que uno de los comandos que había dejado ejecutando ha terminado.

En el futuro, si te preguntas si se te ha olvidado en tu programa cerrar algún descriptor, podrías incluir un flag que haga que tu programa duerma al final y luego utilizar *lsof(1)* para inspeccionarlo.

4. Permisos y control de acceso

Resulta instructivo considerar los permisos y las comprobaciones que hace UNIX para intentar asegurar el acceso a ficheros. Como sabes, cada fichero tiene una *lista de control de acceso*, implementada en un único entero donde cada bit indica un permiso para el dueño, el grupo de usuarios al que pertenece el fichero o el resto del mundo.

Pues bien, esta lista se comprueba *durante open*. En ningún caso `read` o `write` comprueban los permisos del fichero en que operan. Se supone que si un proceso ha tenido permisos para abrir un fichero para escribir en el, por ejemplo, es legítimo que `write` pueda proceder para dicho fichero desde ese proceso.

Las **listas de control de acceso** (ACL en inglés) son similares a los vigilantes de seguridad en la puerta de una fiesta. En este caso el vigilante es UNIX y comprueba para cada proceso (visitante de la fiesta) si puede o no acceder a la misma (al fichero). Una alternativa a una lista de control de acceso es utilizar algo similar a una "llave" que permite la entrada. En este caso, esa "llave" suele denominarse **capability** e indica que quien la posee puede hacer algo con algún recurso (abrir una puerta en el ejemplo).

Aunque los ficheros están protegidos con una ACL, `read`, `write` y el resto de operaciones sobre un fichero abierto operan utilizando el descriptor como *capability*. Una vez tienes el descriptor abierto para

(c) 2016 Francisco J. Ballesteros

escribir puedes escribir en el fichero. Ya no es preciso volver a comprobar los permisos.

La estructura de datos a la que apunta un descriptor de fichero, que veremos más adelante, contiene un campo que registra para qué se abrió el fichero (leer, escribir, leer y escribir) y posteriormente `read` y `write` tan sólo han de comprobar si el descriptor es válido para ellos.

¡Curiosamente el uso de ficheros en UNIX combina tanto ACLs como capabilities!

5. Offsets

Hay una pregunta que seguramente te has hecho. ¿Cómo saben `read` y `write` en qué posición del fichero han de trabajar? Esto es, ¿En qué *offset* debe escribir `write`? o ¿Desde qué *offset* debe leer `read`?

La respuesta procede de `open`. Cuando se abre un fichero, el sistema mantiene la pista de en qué posición del fichero se está trabajando. Inicialmente este offset es cero. Cuando se escribe, se escribe en el offset para el fichero abierto y un efecto lateral de escribir *n* bytes es que el offset aumenta en *n* unidades. Igual sucede en `read`. Cuando se lee, se lee desde el offset que UNIX mantiene para el fichero abierto. Si se leen *n* bytes, el offset aumenta en *n*. Dicho de otro modo, los fichero se leen y escriben **secuencialmente** utilizando `read` y `write`. Si vuelves a mirar la salida de *lsof* que mostramos anteriormente, verás que una de las columnas indica cuál es el offset para cada descriptor.

Por ejemplo, considera este programa:

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    int    fd, i, n;

    fd = open("afile", O_WRONLY|O_CREAT, 0644);
    if (fd < 0) {
        err(1, "afile");
    }
    for (i = 1; i < argc; i++) {
        n = strlen(argv[i]);
        if (write(fd, argv[i], n) != n) {
            close(fd);
            err(1, "write");
        }
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

El programa escribe sus argumentos en el fichero `afile`. La llamada a `open` abre el fichero `afile` para escribir en el (`O_WRONLY`). Como verás, el segundo argumento son flags en un único entero. Se utiliza un *or* para activar los bits que se desean en el entero. `O_CREAT` indica que si no existe queremos crear el

(c) 2016 Francisco J. Ballesteros

fichero. En dicho caso, cuando el fichero no existe y se crea, el tercer argumento indica qué permisos queremos para el nuevo fichero.

Vamos a ejecutarlo por primera vez tras comprobar que `afile` no existe.

```
unix$ ls -l afile
ls: afile: No such file or directory
unix$ writef aa bb cc
```

Si inspeccionamos `afile` veremos que tiene 9 bytes

```
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  9 Aug 20 15:49 afile
```

y que su contenido son los argumentos de `writef` (con un fin de línea añadido tras cada uno):

```
unix$ cat afile
aa
bb
cc
```

Podemos utilizar *xd(1)* para ver los bytes y caracteres que contiene el fichero:

```
unix$ xd -b -c afile
0000000  61 61 0a 62 62 0a 63 63 0a
      0   a  a \n  b  b \n  c  c \n
0000009
```

La primera columna indica el offset en que aparecen los bytes que *xd* muestra a continuación. Los fines de línea son bytes con el valor `0xa` (`10` en decimal). Como verás, hemos hecho seis writes y se han escrito secuencialmente en el fichero. Esto quiere decir que el primer `write` ha escrito en la posición `0` del fichero: ha utilizado el offset `0`. El segundo `write` ha escrito un `"\n"` tras escribir el primer argumento, y ha escrito a partir del offset `2` en el fichero. El tercer `write` ha escrito el segundo argumento en la posición `3`. Y así sucesivamente.

Por cierto, el tamaño del fichero es `9` puesto que el mayor offset escrito ha sido el `8` y empezamos a contar en `0`. Luego el fichero tiene `9` bytes escritos en total. Más allá no hemos escrito nunca. Dicho de otro modo, el tamaño del fichero está determinado por hasta dónde hemos escrito en el fichero.

Si ejecutamos el programa una segunda vez, veremos mejor qué supone utilizar un offset.

```
unix$ writef x y
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  9 Aug 20 15:49 afile
unix$ cat afile
x
y
b
cc
unix$ xd -b -c afile
0000000  78 0a 79 0a 62 0a 63 63 0a
      0   x \n  y \n  b \n  c  c \n
0000009
```

Lo primero que vemos es que ¡el fichero sigue teniendo 9 bytes!. Dicho de otro modo, aunque hemos escrito más veces, el tamaño no ha aumentado. El primer argumento se ha escrito al principio, lo que quiere decir que el primer `write` ha utilizado un offset `0` y ha escrito un byte. El segundo `write` ha escrito el fin

de línea en el offset `1`, dado que el primer write escribió un byte. Etcétera.

Otro detalle curioso es que después del texto escrito ("`x\ny\n`") puedes ver que el fichero contiene "`b\ncc\n`". Lo único que sucede es que esos bytes se escribieron la vez anterior que ejecutamos `writef` y esta vez no los hemos vuelto a escribir, luego tienen el mismo valor que tenían. Recuerda que `write` **no** inserta, `write` (re)escribe.

Y una cosa más. Ahora el fichero tiene 4 líneas, pero no obstante su tamaño sigue siendo 9 bytes. Aunque tenga más líneas, el fichero no es mas grande que antes. Que tenga más líneas se debe simplemente a que hay más "`\n`" escritos en el fichero.

Para ver estas cosas te puede resultar cómodo utilizar *wc(1)*, que cuenta líneas, palabras y caracteres en un fichero, o en la entrada si no indicas fichero alguno.

```
unix$ wc afile
       4       4       9 afile
unix$
```

¿Dónde está el offset? La figura 12 muestra qué aspecto tiene la estructura de datos que usa UNIX.

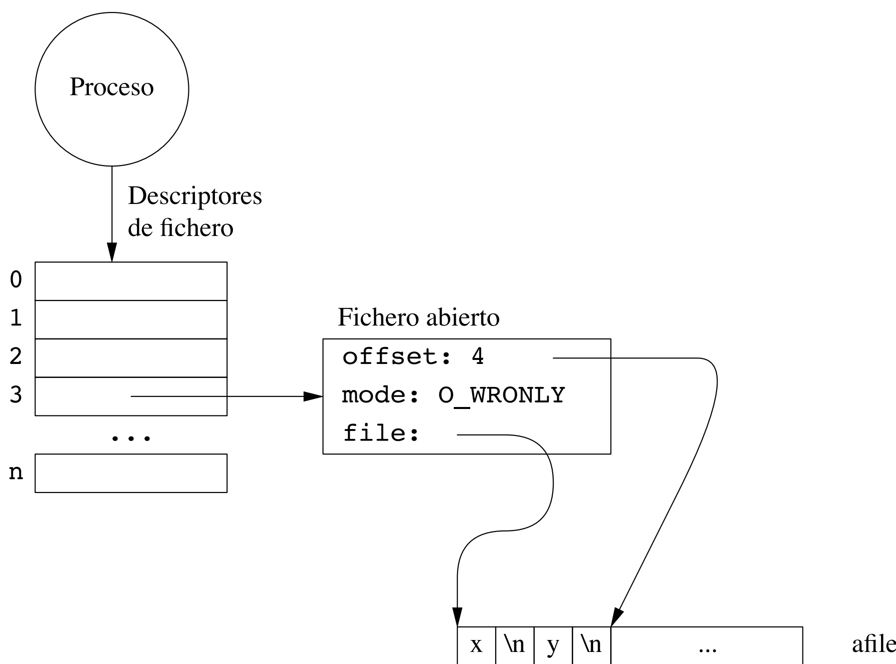

Figura 12: *El offset para lecturas y escrituras se guarda fuera del descriptor de fichero, en una entrada en la tabla de ficheros abiertos.*

En cada descriptor de fichero hay un puntero hacia un record (como siempre). En dicho record, llamado *fichero abierto* por UNIX y almacenado en una *tabla de ficheros abiertos* se guarda:

- El offset en que hay que leer/escribir.
- El modo en que se abrió el fichero (lectura, escritura, o lectura/escritura).
- El puntero hacia la estructura de datos que representa el fichero en UNIX.

En la figura se ve qué valor tenía el contenido de `afile` y el `offset` del fichero abierto justo al final de ejecutar por segunda vez `afile`, antes de llamar a `close`.

(c) 2016 Francisco J. Ballesteros

Si queremos que cada vez que ejecutemos `writef` el fichero `afile` quede justo con lo que hemos escrito, la solución es eliminar el contenido de `afile` cuando el fichero ya existe, durante `open`. Esto es, si el fichero no existe lo creamos y si existe lo *truncamos* a 0-bytes. El flag `O_TRUNC` de `open` consigue este efecto. Basta utilizar

```
fd = open("afile", O_WRONLY|O_TRUNC|O_CREAT, 0644);
```

en lugar de la llamada a `open` que hacíamos antes. La próxima vez que ejecutemos `writef` no quedarán restos del contenido que `afile` pudiera tener antes si es que existía. Aunque claro, si has leído *open(2)* ya lo sabías.

Existe otro flag que podemos utilizar con *open(2)* que afecta al offset que utiliza `write`. Se trata del flag `O_APPEND`. Si indicamos `O_APPEND` en `open`, las llamadas a `write` ignoran el offset del fichero y escriben siempre al final del mismo. En realidad, se escribe justo en la posición indicada por el tamaño del fichero. Pero cuidado aquí si el fichero es un fichero compartido en red: cada UNIX que tiene abierto el fichero podría tener su propia idea del tamaño del mismo (por ejemplo si un UNIX acaba justo de hacer crecer el fichero pero otro no se ha dado cuenta todavía).

6. Ajustando el offset

Para evitar errores utilizando `read` y `write` basta con que pienses que UNIX es muy obediente y hace justo lo que se supone que cada llama hace. No hay magia. Si recuerdas lo que hemos visto, podrás entender lo que ocurre. ¡Vamos a comprobarlo!

Hay otra llamada, *lseek(2)*, que permite cambiar el offset de un descriptor de fichero. Bueno, en realidad, queremos decir "*el offset de un fichero abierto al que apunta un descriptor*". Se le suministra el descriptor de fichero, cuantos bytes queremos mover el offset y desde donde contamos. Por ejemplo,

- `lseek(fd, 10, SEEK_SET)` fija el offset a 10 en `fd`.
- `lseek(fd, 10, SEEK_CUR)` añade 10 al offset de `fd`.
- `lseek(fd, 10, SEEK_END)` fija el offset 10 bytes contando desde el final del fichero hacia atrás en `fd`.

Las constantes `SEEK_SET`, `SEEK_CUR` y `SEEK_END` son simplemente `0`, `1` y `2`. De hecho, dado que desde el comienzo de UNIX han tenido esos valores, es muy poco probable que cambien nunca de valor.

Entendido esto, fíjate en este programa:

(c) 2016 Francisco J. Ballesteros

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>

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

    fd = open("afile", O_WRONLY|O_TRUNC|O_CREAT);
    if (fd < 0) {
        err(1, "open: afile");
    }
    lseek(fd, 32, 0);
    if (write(fd, "xx\n", 3) != 3) {
        close(fd);
        err(1, "write");
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Ejecutándolo podemos ver lo que sucede con el nuevo `afile` en este caso.

```
unix$ seekwrite
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  35 Aug 20 16:32 afile
unix$ cat afile
xx
unix$ xd -b -c afile
0000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000020  78 78 0a
     20   x  x \n
0000023
```

¡Curioso!, ¿No? Resulta que *ls* dice que el fichero tiene 35 bytes. Pero sólo hemos escrito 3 bytes. Veamos...

- `lseek` ha cambiado el offset a 32, contando desde el comienzo del fichero.
- `write` ha escrito tres bytes en dicho offset.

Luego en total, el byte más alto que hemos escrito es el 34 (contando desde 0), lo que quiere decir que mirando los 35 primeros bytes tenemos el contenido del fichero que hemos escrito alguna vez. Además, puedes ver que los bytes que no hemos escrito nunca están a cero. Eso es cortesía de UNIX.

El otro detalle curioso es que cuando hemos hecho un *cat* del fichero sólo hemos visto una línea con `xx`, que era lo que cabría esperar. Pero en realidad, sí que `cat` ha escrito un total de 35 bytes en su salida:

(c) 2016 Francisco J. Ballesteros

```
unix$ cat afile >/tmp/out
unix$ ls -l /tmp/out
-rw-r--r--  1 nemo  wheel  35 Aug 20 16:37 /tmp/out
unix$
```

Lo que ocurre es que cuando *cat* escribe un byte a cero en su salida estándar, y esta es el terminal, el terminal no sabe cómo mostrar dicho byte (¿Cómo dibujar un símbolo para el byte nulo?) y no lo escribe en absoluto. Es normal, `/dev/tty` sólo espera que escribas texto en el teclado y que muestres texto en la pantalla. No espera efectos especiales.

Utilizar `lseek` como hemos hecho es muy habitual. Por ejemplo, para crear un fichero con 1GiB basta con que hagas un `lseek` a la posición `1024*1024*1024-1` y escribas un byte. El tamaño del nuevo fichero será justo de 1GiB. Lo que es más, es muy posible que UNIX no asigne espacio en disco para el nuevo fichero salvo para el último *bloque* que use el fichero (que es dónde has escrito el byte). Todos los trozos anteriores del fichero están sin escribir y, si se leen, UNIX sabe que se leen con todos los bytes a cero. Siendo esto así, ¿Para qué guardar los ceros en el disco si UNIX sabe que están a cero? A estos ficheros se les llama *ficheros con huecos*. ¡Puedes tener ficheros más grandes que el tamaño del disco en que los guardas!

Otro uso de `lseek` es para obtener el offset. Basta con cambiar el offset a 0 bytes contando a partir del offset actual, y utilizar el valor que retorna `lseek` (que es el nuevo offset). Por ejemplo:

```
off = lseek(fd, 0, 1);
```

7. Crear y borrar

Ya hemos creado ficheros utilizando el flag `O_CREAT` de `open`. Otra forma es utilizar *creat(2)*. Llamar a

```
fd = creat(path, mode);
```

es lo mismo que llamar a

```
fd = open(path, O_CREAT|OTRUNC|O_WRONLY, mode);
```

así que en realidad ya sabes utilizar `creat`.

No es posible crear directorios utilizando `creat`. Hay que utilizar *mkdir(2)*. Su uso es similar al de `creat`, salvo porque `mkdir` no devuelve ningún descriptor. Tan sólo devuelve `-1` si falla y `0` en caso contrario:

```
if (mkdir("/tmp/mydir", 0755) < 0) {
    ...mkdir ha fallado...
}
```

Los permisos (o como UNIX lo denomina, el *modo*) con que se crean los ficheros y directorios no sólo dependen de los permisos que indiques en la llamada a `open`, `creat` o `mkdir`. Por seguridad, cada proceso tiene un atributo denominado *umask*. El *umask* es la *máscara de creación de ficheros* y actúa como una máscara para evitar que permisos no deseados se den a ficheros que crea un proceso. La llamada al sistema es *umask(2)* y el comando que utiliza dicha llamada y podemos utilizar en el shell es *umask(1)*.

Para cambiar la *umask* desde C podemos utilizar código como

```
umask(077);
```

Con esta llamada ponemos la máscara a 0077, lo que hace que en ningún caso se den permisos de lectura, escritura o ejecución para el grupo de usuarios o para el resto del mundo. Si con esta máscara utilizamos 0755 como modo en `creat`, los permisos del nuevo fichero serán en realidad 0700. Aunque no hemos

(c) 2016 Francisco J. Ballesteros

usado el valor que devuelve `umask`, la llamada devuelve la máscara anterior, por si queremos volverla a dejar como estaba.

Normalmente, la máscara es 0022, lo que hace que ni el grupo ni el resto del mundo pueda escribir los ficheros o directorios que creamos. ¿Puedes ahora explicar lo que sucede en esta sesión de shell?

```
unix$ umask
0022
unix$ touch a
unix$ ls -l a
-rw-r--r--  1 nemo  staff  0 Aug 20 21:32 a
unix$ umask 077
unix$ umask
0077
unix$ rm a
unix$ touch a
unix$ ls -l a
-rw-------  1 nemo  staff  0 Aug 20 21:32 a
unix$
```

Cuando un proceso crea un fichero hace en realidad dos cosas:

- crear el fichero
- darle un nombre en un directorio

Ambas cosas suceden dentro de la llamada. Esto hace que resulte natural que no exista una llamada para borrar un fichero. Para borrar un fichero lo que se hace es eliminar el nombre del fichero. Si nadie está utilizando el fichero y no hay ningún otro nombre para el fichero, este se borra.

Por ejemplo, este programa es una versión simplificada de *rm(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int i, sts;

    sts = 0;
    if (argc == 1) {
        fprintf(stderr, "usage: %s file...\n", argv[0]);
        exit(1);
    }
    for (i = 1; i < argc; i++) {
        if (unlink(argv[i]) < 0) {
            warn("%s: unlink", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

(c) 2016 Francisco J. Ballesteros

Podemos utilizarlo para borrar ficheros:

```
unix$ touch /tmp/a /tmp/b /tmp/c
unix$ ls -l /tmp/?
-rw-r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/a
-rw-r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/b
-rw-r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/c
unix$ chmod -w /tmp/b
unix$ ls -l /tmp/b
-r--r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/b
unix$ rem /tmp/[abc]
unix$ ls -l /tmp/?
ls: /tmp/?: No such file or directory
unix$
```

Hemos utilizado `/tmp/?` para que el shell escriba por nosotros en la línea de comandos todos los nombres de fichero que están en `/tmp` y tienen como nombre un sólo carácter. También hemos utilizado `/tmp/[abc]` para que el shell escriba por nosotros los nombres de ficheros en `/tmp` que sean `a`, `b` o `c`. Si esto te resulta confuso, piensa que hemos usando siempre los argumentos que hemos dado a `touch` en lugar de `/tmp/[abc]` o de `/tmp/?`. Más adelante explicaremos estas expresiones con más detalle.

Como verás, que no tengas permiso de escritura en un fichero no quiere decir que no puedas borrarlo. Pero mira esto:

```
unix$ mkdir /tmp/d
unix$ touch /tmp/d/a
unix$ chmod -w /tmp/d
unix$ 8.rem /tmp/d/a
rem: /tmp/d/a: unlink: Permission denied
unix$ chmod +w /tmp/d
unix$ rem /tmp/d/a
unix$
```

Como indica la página de manual *unlink(2)*, borrar un fichero requiere poder escribir el directorio en que está. Cuando tengas dudas respecto a permisos, consulta el manual.

Otra cosa curiosa sucede si intentamos borrar `/tmp/d`.

```
unix$ rem /tmp/d
rem: /tmp/d: unlink: Operation not permitted
unix$
```

La llamada `unlink` no sabe borrar directorios.

Para borrar un directorio hay que utilizar *rmdir(2)*, del mismo modo que para crear directorios hay que utilizar *mkdir(2)* y no podemos utilizar *creat(2)*.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int i, sts;

    sts = 0;
    if (argc == 1) {
        fprintf(stderr, "usage: %s dir...\n", argv[0]);
        exit(1);
    }
    for (i = 1; i < argc; i++) {
        if (rmdir(argv[i]) < 0) {
            warn("%s: rmdir", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Y ahora podemos...

```
unix$ remdir /tmp/d
unix$
```

Naturalmente, no podemos utilizar `rmdir` para borrar ficheros:

```
unix$ touch /tmp/a
unix$ remdir /tmp/a
remdir: /tmp/a: rmdir: Not a directory
unix$
```

¡Y tampoco para borrar directorios que no están vacíos! (sin contar ni "`.`" ni "`..`"). Si `rmdir` pudiese borrar directorios no vacíos nos divertiríamos mucho ejecutando

```
unix$ rm /
```

8. Enlaces

Dado un fichero que existe, podemos darle un nuevo nombre dentro de la misma partición o del mismo sistema de ficheros. Esto se hace con la llamada *link(2)*. Este programa es similar al comando *ln(1)*, que establece un nuevo nombre para un fichero:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (link(argv[1], argv[2]) < 0) {
        err(1, "link %s", argv[1]);
    }
    exit(0);
}
```

Vamos a verlo despacio y paso a paso utilizando un par de ficheros. Primero creamos un fichero `afile`:

```
unix$ echo hola >afile
unix$ ls -i afile
7782892 afile
unix$ cat afile
hola
```

Hemos utilizado `ls` con la opción `-i` para que nos de el número que usa UNIX para identificar el fichero dentro su partición o sistema de ficheros (UNIX lo llama *i-nodo*).

Ahora podemos ejecutar nuestro programa `lnk` o el comando `ln` (funcionan igual en este caso) para dar un nuevo nombre para `afile`:

```
unix$ lnk afile another
unix$ ln afile another
ln: another: File exists
unix$
```

La segunda vez, el fichero `another` ya existe por lo que no se puede utilizar ese nombre como un nuevo nombre. En cualquier caso, `lnk` ha llamado a `link` haciendo que `another` sea otro nombre para `afile`. ¡Ambos ficheros son el mismo!

Aunque en este caso ambos nombres están dentro del mismo directorio, es posible crearlos en directorios distintos. Pero sigamos explorando los enlaces y el uso que hace UNIX de los nombres. Primero, podemos comprobar que el fichero es en realidad el mismo:

```
unix$ ls -i another
7782892 another
```

El número de i-nodo sólo significa algo dentro de la misma partición en el mismo disco, pero ese es el caso por lo que si el número coincide entonces el fichero es el mismo.

Veámoslo mirando y cambiando uno de los ficheros:

```
unix$ cat another
hola
unix$ echo adios >afile
unix$ cat another
adios
```

Tras cambiar `afile`, resulta que `another` ha cambiado del mismo modo. ¡Naturalmente!, son el mismo fichero.

Si borramos `afile`, todavía sigue existiendo el fichero

```
unix$ rm afile
unix$ cat another
adios
```

puesto que aún tiene otro nombre. Si borramos el único nombre que le queda al fichero

```
unix$ rm another
```

UNIX borra realmente el fichero.

Para saber cuántos nombres tiene un fichero, UNIX guarda en la estructura de datos que lo implementa (llamada *i-nodo*) un contador que cuenta cuántos nombres tiene. Se lo suele llamar contador de referencia. Podemos verlo utilizando `ls`. Fíjate en el número de la segunda columna cada vez que llamamos a `ls`:

```
unix$ touch afile
unix$ ln afile another
unix$ ls -l afile another
-rw-r--r--  2 nemo  staff  0 Aug 20 19:20 afile
-rw-r--r--  2 nemo  staff  0 Aug 20 19:20 another
unix$ rm another
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  0 Aug 20 19:20 afile
```

¿Recuerdas que "`.`" es otro nombre para el directorio actual? Observa la salida de este comando:

```
unix$ ls -ld .
drwxr-xr-x  2 nemo  staff  1258 Aug 20 19:21 .
```

El flag `-d` de `ls` hace que si listamos un nombre de directorio, `ls` liste el fichero del directorio y no los ficheros que contiene. ¿Puedes explicar por qué "`.`" tiene 2 enlaces?

Podemos jugar más...

```
unix$ mkdir /tmp/d
unix$ mkdir /tmp/d/1 /tmp/d/2 /tmp/d3
unix$ ls -ld /tmp/d
drwxr-xr-x  4 nemo  wheel  136 Aug 20 19:26 /tmp/d
```

¿Aún no ves por qué "`/tmp/d`" tiene 4 enlaces? Aquí tienes una pista...

```
unix$ ls -ld /tmp/d/1/..
drwxr-xr-x  4 nemo  wheel  136 Aug 20 19:26 /tmp/d/1/..
```

9. Lectura de directorios

Los directorios en UNIX son ficheros que contienen una tabla de ficheros pertenecientes al directorio. En cada entrada de la tabla hay dos cosas:

(c) 2016 Francisco J. Ballesteros

- Un nombre de fichero
- Un número (de *i-nodo*) que identifica al fichero.

Cuando se crea un fichero con `creat`, `open`, o `mkdir` se añade automáticamente la entrada de directorio para el fichero o directorio que se ha creado. Recordarás que `link` (o `ln`) puede usarse para establecer nuevos enlaces para ficheros que existen (nuevas entradas de directorio con el número de un fichero existente) y que `unlink` o `rmdir` pueden utilizarse para eliminar nombres.

Hace tiempo UNIX permitía utilizar `read` y `write` para leer y escribir directorios. No obstante, esto causó tantos problemas cada vez un usuario escribía incorrectamente la tabla que desde hace tiempo no se puede utilizar `read` o `write` para leer una tabla de directorio.

En lugar de *read(2)*, para directorios tenemos la llamada al sistema *getdirentries(2)*. No obstante, es mucho más fácil utilizar las funciones *opendir(3)*, *readdir(3)* y *closedir(3)* que tenemos en la librería de C. No sólo es más fácil. Es más portable. Piensa que el formato exacto de una entrada de directorio depende del tipo de sistema de ficheros que utilices (del programa que implementa los ficheros en esa partición o donde quiera que estén los ficheros, y que forma parte del kernel).

Aquí tienes un ejemplo de cómo listar un directorio en C.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/dir.h>
#include <dirent.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    DIR *d;
    struct dirent *de;

    d = opendir(".");
    if(d == NULL) {
        err(1, "opendir");
    }
    while((de = readdir(d)) != NULL) {
        printf("%s\n", de->d_name);
    }
    if (closedir(d) < 0) {
        err(1, "closedir");
    }
    exit(0);
}
```

Este programa lista el contenido del directorio actual. Por ejemplo,

```
unix$ lsdot
.
..
lsdot.c
lsdot
ch03e.w
unix$
```

Como verás, aparecen tanto "`.`" como "`..`".

(c) 2016 Francisco J. Ballesteros

Cada llamada a `readdir` devuelve una estructura de tipo `struct dirent`. Dicha estructura no debes liberarla, según indica el manual, y si llamas de nuevo a `readdir` la estructura antigua se sobreescribe con la nueva muy posiblemente.

Esta estructura cambia mucho de un tipo de UNIX a otro, aunque en la mayoría las entradas de directorio son similares. Por ejemplo, esta es la que utiliza Linux:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* not an offset; see NOTES */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* filename */
};
```

Normalmente, siempre tienes el campo `d_name`. En Linux y sistemas BSD tienes `d_type` también. El resto de campos suelen variar de nombre y tal vez no estén en tu UNIX. Si utilizas algo más que `d_name`, seguramente tu código deje de ser portable a otros tipos de UNIX.

Cuando tienes `d_type`, este campo vale `DT_DIR` para directorios, `DT_REG` para ficheros (ficheros regulares o normales) y otros valores para otros tipos de fichero que veremos más adelante.

Naturalmente, nunca se escribe un directorio, este se actualiza creando, borrando y renombrando los ficheros que contiene. Ya conoces todo lo necesario, salvo por cómo renombrar ficheros. La llamada al sistema en cuestión es *rename(2)*.

Este programa renombra un fichero, similar al uso más sencillo del comando *mv(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

int
main(int argc, char* argv[])
{

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (rename(argv[1], argv[2]) < 0) {
        err(1, "rename %s", argv[1]);
    }
    exit(0);
}
```

Podemos usarlo para mover un fichero a notro nombre:

(c) 2016 Francisco J. Ballesteros

```
unix$ echo hola >a
unix$ cat a
hola
unix$ mvf a b
unix$ ls -l a b
ls: a: No such file or directory
-rw-------  1 nemo  staff  5 Aug 20 22:13 b
unix$ cat b
hola
unix$
```

Has de tener en cuenta que si el fichero de destino existe, se borra (su nombre) durante el *rename*. Además, es un error hacer un *rename* hacia un directorio no vacío. Y otra cosa, si recuerdas el comando *mv(1)*, verás que

```
unix$ mv a /tmp
```

mueve el fichero `a` a `/tmp/a`. No obstante,

```
unix$ mvf a /tmp
mvf: rename a: Permission denied
```

nuestro programa no es tan listo e intenta hacer que `a` tenga como nombre `/tmp` (que es un directorio que ya existe).

10. Globbing

Dado que gran parte de las "palabras" que escribimos en una línea de comandos corresponden a nombres de fichero, el shell da facilidades para expresar de un modo compacto nombres de fichero que tienen cierto aspecto. Dicho de otro modo, el shell permite utilizar expresiones que *generan* nombres de fichero o que *encajan* con nombres de fichero. A esta herramienta se la conoce como *globbing*.

Ya sabes que en general una buena forma de saber cómo funcionan las cosas es experimentar con ellas, así que vamos a crear un directorio con ciertos ficheros dentro para experimentar, tal y como harías tu.

```
unix$ mkdir -p /tmp/d/d2/d3
unix$ touch /tmp/d/a /tmp/d/ab /tmp/d/abc
unix$ touch /tmp/d/d2/j /tmp/d/d2/k
unix$ touch /tmp/d/d2/d3/x  /tmp/d/d2/d3/y
unix$
```

El flag `-p` de `mkdir` hace que se cree el directorio indicado como argumento y los directorios padre si es que no existen.

Podemos utilizar el comando *du(1)* (*disk usage*) para que liste el árbol de ficheros que hemos creado. Este comando en realidad lista cuándo disco consumen ficheros y directorios, pero es una forma práctica de listar árboles de ficheros (¡Más práctica que utilizando `ls`!).

(c) 2016 Francisco J. Ballesteros

```
unix$ du -a /tmp/d
0	/tmp/d/1
0	/tmp/d/2
0	/tmp/d/a
0	/tmp/d/ab
0	/tmp/d/abc
0	/tmp/d/d2/d3/x
0	/tmp/d/d2/d3/y
0	/tmp/d/d2/d3
0	/tmp/d/d2/j
0	/tmp/d/d2/k
0	/tmp/d/d2
0	/tmp/d
```

El flag `-a` de `du` hace que liste no sólo los directorios, sino también los ficheros.

Pero continuemos. Cuando el shell ve que un nombre en la línea de comandos contiene ciertos caracteres especiales, intenta encontrar paths de fichero que encajan con dicho nombre y, si los encuentra, cambia ese nombre los pos paths, separados por espacio. En realidad, deberíamos llamar *expresión* a lo que estamos llamando "nombre".

Por ejemplo, en este comando

```
unix$ echo /tmp/d/*
/tmp/d/1 /tmp/d/2 /tmp/d/a /tmp/d/ab /tmp/d/abc /tmp/d/d2
```

el shell ha tomado "`/tmp/d/*`" y, como puedes ver por la salida de `echo`, lo ha cambiado por los nombres de fichero en "`/tmp/d`".

Pero mira este otro comando:

```
unix$ echo /tmp/d/a*
/tmp/d/a /tmp/d/ab /tmp/d/abc
```

Aquí el shell ha cambiado "`/tmp/d/a*`" por los nombres de fichero en `/tmp/d` que comienzan por "`a`".

Veamos otro más...

```
unix$ echo /*/ls
/bin/ls
```

El shell ha buscado en "`/`" cualquier fichero que sea un directorio y contenga un fichero llamado "`ls`".

Las expresiones de globbing son fáciles de entender:

- El shell las recorre componente a componente mirando cada una de las subexpresiones que corresponden a nombres de directorio en el path.
- Naturalmente, "`/`" separa unos componentes de otros.
- En cada componente intenta encontrar todos los ficheros cuyo nombre encaja en la subexpresión para dicho componente.
- En cada componente podemos tener caracteres normales o alguna de estas expresiones:
  - "`*`" encaja con cualquier string, incluso con el string vacío.
  - "`[...]`" encaja con cualquier carácter contenido en los corchetes. Aquí es posible indicar rangos de caracteres como en "`[a-z]`" (las letras de la "`a`" a la "`z`").
  - "`?`" encaja con un sólo carácter.

Por ejemplo,

(c) 2016 Francisco J. Ballesteros

```
unix$ cd /tmp/d
unix$ ls
1    2    a    ab    abc    d2
unix$ echo [12]*
1 2
unix$ echo [ab]*
a ab abc
unix$
```

La primera expresión quiere decir: "un `1` o un `2` y luego cualquier cosa" La segunda quiere decir: "una `a` o una `b` y luego cualquier cosa".

Otro ejemplo:

```
unix$ ls /tmp/*/d*/?
/tmp/d/d2/j    /tmp/d/d2/k
```

Esto es, dentro de "`/tmp`", cualquier nombre de directorio que tenga dentro un directorio que comience por "`d`" y tenga dentro un fichero cuyo nombre sea un sólo carácter y nada más.

Quizá el uso más común sea en comandos como

```
unix$ ls *.[ch]
```

para listar los fuentes en C (ficheros cuyo nombre es cualquier cosa, luego un "`.`" y luego o bien una "`c`" o una "`h`", y como

```
unix$ rm *.o
```

para borrar todos los ficheros objeto (cualquier nombre terminado en "`.o`").

Recuerda que ninguno de estos comandos sabe nada respecto a globbing, ni entienden qué quiere decir "`*`" ni lo ven siquiera. Es el shell cuando analiza la línea de comandos el que detecta que hay una expresión que contiene caracteres de globbing, y cambia dicha expresión por los nombres de los ficheros que encajan en la expresión. Los comandos simplemente reciben los argumentos con los paths y se limitan a hacer su trabajo.

En ocasiones resultará útil hacer que el shell no haga globbing, como en esta sesión:

```
unix$ touch '*'
```

Hemos creado un fichero llamado "`*`". Esta vez a sabiendas, pero podría haber sido por error. Para hacerlo simplemente utilizamos las comillas simples para hacer que el shell tome *literalmente* y sin cambiar nada lo que incluyen las comillas (y como una sóla palabra).

Y ahora tenemos el dilema.

```
unix$ ls
*    1    2    a    ab    abc    d2
unix$ echo *
* 1 2 a ab abc d2
```

Si utilizamos

```
unix$ rm *
```

para eliminar "`*`", ¡borraremos todos los ficheros! y no es lo que queremos. Pero podemos hacer esto:

(c) 2016 Francisco J. Ballesteros

```
unix$ rm '*'
unix$ ls
1    2    a    ab    abc    d2
```

11. Metadatos

Los ficheros, como ya sabes en este punto, no sólo tienen datos. Tienen además atributos, o datos sobre los datos, o **metadatos**. Por ejemplo, el *uid* del dueño del fichero, el *gid* del grupo al que pertenece, los permisos, el contador de referencias, etc.

En UNIX, todos estos metadatos se guardan en la estructura de datos que representa al fichero, que es un record que UNIX llama *i-nodo*. La llamada al sistema *stat(2)* sirve para recuperar los metadatos de un fichero. El siguiente programa mejora nuestro programa para listar directorios y muestra distintos atributos de cada fichero contenido en los directorios cuyo nombre se indica como argumento (el directorio actual si no se indica ninguno).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <time.h>
#include <err.h>
typedef unsigned long long uvlong_t;
static int
list(char *fname)
{
    struct stat st;

    if (stat(fname, &st) < 0)
        return -1;
    printf("%s:\n\t", fname);
    if ((st.st_mode & S_IFMT) == S_IFDIR)
        printf("dir");
    else if ((st.st_mode & S_IFMT) == S_IFREG)
        printf("file");
    else
        printf("type %xu", st.st_mode&S_IFMT);
    printf(" perms %o", st.st_mode & 0777);
    printf(" sz %llu", (uvlong_t) st.st_size);
    printf(" uid %d gid %d", st.st_uid, st.st_gid);
    printf("\n\tlinks %d", (int)st.st_nlink);
    printf(" dev %d ino %llu\n\t", (int)st.st_dev, (uvlong_t) st.st_ino);
    printf("mtime %llus = %s", (uvlong_t)st.st_mtime, ctime(&st.st_mtime));
    return 0;
}
```

El propósito de `list` es listar los atributos de fichero cuyo path se indica en `fname`. La llamada a `stat` rellena una estructura `st` con dichos atributos. Igual que sucedía con la lectura de directorios, el contenido de esta estructura varía de unos sistemas UNIX a otros. Así que hay que tener cuidado para que el código sea portable. Los campos que utilizamos en nuestro programa suelen estar disponibles en todos los UNIX. En el caso de Linux, la estructura está definida así:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

El campo `st_mode` contiene tanto los permisos como otros bits que indican el tipo de fichero. La constante `S_IFMT` indica qué bits contienen el tipo de fichero. Además, tenemos constantes `S_IFDIR`, `S_IFREG`, etc. para comprobar qué tipo de fichero tenemos. Observa que utilizamos "`&`" dado que el campo `st_mode` tiene más bits y no es posible utilizar sólo "`==`".

Otros campos como `st_size`, `st_uid` y `st_gid` debieran ser obvios a estas alturas. El campo `st_nlink` indica cuántas referencias (nombres) tiene el fichero.

Para ver si dos ficheros son el mismo habría que comprobar si tanto `st_dev` como `st_ino` coinciden. El primero indica en qué dispositivo se encuentra el fichero y el segundo indica qué número de fichero dentro de dicho dispositivo (qué número de *i-nodo*) tenemos.

Los campos `st_mtime`, `st_atime` y `st_ctime` contienen las fechas de la última modificación del fichero, del último acceso a los datos del mismo y de la última vez que se cambiaron los atributos (o de cuándo se creó el fichero si no se han cambiado). Estas fechas están codificadas como un número de segundos desde una fecha data, pero podemos utilizar *ctime(3)* para convertir dicho número a un string con la fecha en un formato que un humano pueda leer. En breve veremos la gestión del tiempo en UNIX más despacio.

Lamentablemente, hoy en día, UNIX utiliza tipos como `ino_t`, `dev_t`, `size_t` y otros muchos en lugar de `int`, `long`, etc. Eso quiere decir que tendremos problemas para utilizar los formatos de `printf` con cada uno de ellos. Una solución portable para imprimir dichos valores es la que puedes ver en el código: convertimos el entero en cuestión a otro entero de mayor o igual tamaño, en nuestro caso a `unsigned long long`, y utilizamos el formato para dicho entero (en nuestro caso "`%llu`").

Habría sido mejor utilizar otro tamaño de entero y usar simplemente "`int`" y "`%d`" dado que hoy día la memoria es barata. Pero esto es tan sólo una opinión.

Para que puedas ver el programa en su conjunto y repasar cómo se leían directorios, comprobaban argumentos y otros detalles, mostramos a continuación el resto del fichero fuente que contiene nuestro programa. Observa cómo las funciones se toman molestias en devolver una indicación de error si tienen problemas y cómo se comprueban los errores e imprimen mensajes de error. En aquellos casos en que es posible continuar el trabajo tras informar de un error, el código hace justo eso. Además, se cierran los ficheros que se abren, tengamos errores o no. Pero antes de verlo, esta es una ejecución del programa:

(c) 2016 Francisco J. Ballesteros

```
unix$ ll
./ll:
    file perms 755 sz 9092 uid 501 gid 20
    links 1 dev 16777220 ino 7791046
    mtime 1471771999s = Sun Aug 21 11:33:19 2016
./guide:
    file perms 644 sz 1396 uid 501 gid 20
    links 1 dev 16777220 ino 7766071
    mtime 1471700478s = Sat Aug 20 15:41:18 2016
./ll.c:
    file perms 644 sz 1731 uid 501 gid 20
    links 1 dev 16777220 ino 7788145
    mtime 1471771995s = Sun Aug 21 11:33:15 2016
./writef.c:
    file perms 644 sz 512 uid 501 gid 20
    links 1 dev 16777220 ino 7782848
    mtime 1471700968s = Sat Aug 20 15:49:28 2016
```

Y este es el código que falta:

```
static int
ldir(char *dir)
{
    DIR *d;
    struct dirent *de;
    char path[1024];
    int n, rc;

    d = opendir(dir);
    if(d == NULL) {
        warn("opendir: %s", dir);
        return -1;
    }
    rc = 0;
    while((de = readdir(d)) != NULL) {
        n = snprintf(path, sizeof path, "%s/%s", dir, de->d_name);
        if (n >= sizeof path-1) { // -1 for '\0'
            warn("path %s/%s too long", dir, de->d_name);
            rc = -1;
        } else if (list(path) < 0) {
            warn("list: %s", path);
            rc = -1;
        }
    }
    if (closedir(d) < 0) {
        warn("closedir: %s", dir);
        return -1;
    }
    return rc;
}
```

(c) 2016 Francisco J. Ballesteros

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

    sts = 0;
    if (argc == 1) {
        if (ldir(".") < 0) {
            sts = 1;
        }
    } else {
        for (i = 1; i < argc; i++) {
            if (ldir(argv[i]) < 0) {
                sts = 1;
            }
        }
    }
    exit(sts);
}
```

Por cierto, en ocasiones tendrás un descriptor de fichero del que desees obtener los metadatos, en lugar de tener el path para dicho fichero. En ese caso puedes utilizar *fstat(2)* en lugar de *stat(2)*, que recibe un descriptor de fichero en lugar de un path. En muchas otras llamadas se sigue el mismo convenio y dispones de funciones que comienzan por `"f"` para utilizar un descriptor en lugar de un path para identificar el fichero con que quieres trabajar. ¡El manual es tu amigo!

12. El tiempo

UNIX mantiene su propia idea de la fecha y hora. Por un lado, el hardware habitualmente dispone de un reloj que está continuamente alimentado por baterías y que se ocupa de mantener la noción del tiempo.

Naturalmente, este reloj es un contador que se incrementa cada unidad de tiempo. Normalmente dispone de una serie de ticks por segundo y es programable respecto a la frecuencia a la que opera.

El sistema además suele programar el temporizador hardware para que cada `HZ` (una constante entera) veces por segundo genere una interrupción de reloj. Esto se usa, como ya mencionamos, entre otras cosas para expulsar del procesador aquellos procesos que llevan suficiente tiempo ejecutando (que han agotado su cuanto). Es tan simple como retornar de la interrupción en un proceso distinto.

Desde el shell, ya sabes que el comando *date(1)* informa respecto a la fecha y hora. Desde C, el principal interfaz para obtener el tiempo es *gettimeofday(2)* (y puede utilizarse *settimeofday(2)* para ajustarlo).

Esta función rellena una estructura de tipo *timeval* con los segundos y microsegundos desde una fecha convenida, llamada *epoch*. Normalmente desde el 1 de enero de 1970 en el caso de UNIX.

```
struct timeval {
        time_t       tv_sec;   /* seconds since Jan. 1, 1970 */
        suseconds_t  tv_usec;  /* and microseconds */
};
```

Además, rellena otra estructura llamada *timezone* que indica la zona horaria en que nos encontramos y si estamos en horario de verano (*daylight saving time*).

(c) 2016 Francisco J. Ballesteros

```
struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

Pero *no deberías* dejar que `gettimeofday` rellene información sobre la zona horaria. Es mejor utilizar funciones de *ctime(3)* si deseas jugar con zonas horarias. Por ello nosotros vamos a utilizar `NULL` en el argumento correspondiente a la zona horaria para que `gettimeofday` lo ignore.

Este programa es similar a *date(1)*, pero con la opción "`-n`" imprime el número de segundos y microsegundos y los datos de la zona horaria en lugar de escribir la fecha normalmente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>
#include <time.h>
#include <sys/time.h>

typedef unsigned long long uvlong_t;
static char *argv0;

static void
usage(void)
{
    fprintf(stderr, "usage: %s [-n]\n", argv0);
    exit(1);
}
```

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    struct timeval tv;
    int nflag;

    nflag = 0;
    argv0 = argv[0];
    if (argc == 2) {
        if (strcmp(argv[1], "-n") == 0) {
            nflag = 1;
        } else {
            usage();
        }
    }
    if (argc > 2) {
        usage();
    }

    if (gettimeofday(&tv, NULL) < 0) {
        err(1, "gettimeofday");
    }
    if (nflag) {
        printf("%llds %lldµs\n", (uvlong_t)tv.tv_sec, (uvlong_t)tv.tv_usec);
    } else {
        printf("%s", ctime(&tv.tv_sec));
    }
    exit(0);
}
```

La función *ctime(3)* se ocupa de generar un string para la fecha dada como un número de segundos desde *epoch*. Todas los argumentos de funciones de *ctime(3)* que aceptan un `time_t` suelen ser dicho número de segundos.

Para partir la fecha dada por un `time_t` (número de segundos desde *epoch*) y obtener el año, el mes, el día del mes, etc. normalmente se utilizan las funciones `localtime` (hora local) y `gmtime` (hora en greenwich). Por ejemplo, como en

```
struct tm *t;
         struct timeval tv;
if (gettimeofday(&tv, NULL) < 0) {
    err(1, "gettimeofday");
}
t = localtime(&tv.tv_sec);
```

o bien

```
t = gmtime(&tv.tv_sec);
```

Y luego podemos usar los siguientes campos de la estructura `tm`:

(c) 2016 Francisco J. Ballesteros

```
int tm_sec;     /* seconds (0 - 60) */
int tm_min;     /* minutes (0 - 59) */
int tm_hour;    /* hours (0 - 23) */
int tm_mday;    /* day of month (1 - 31) */
int tm_mon;     /* month of year (0 - 11) */
int tm_year;    /* year - 1900 */
int tm_wday;    /* day of week (Sunday = 0) */
int tm_yday;    /* day of year (0 - 365) */
int tm_isdst;   /* is summer time in effect? */
```

Podríamos tener otros campos dependiendo del UNIX que usemos, pero seguramente no estén disponibles en todos los sistemas.

La función `mktime`, también documentada en *ctime(3)* hace el proceso inverso y genera un tiempo en segundos desde *epoch* a partir de una estructura de tipo `tm`.

13. Cambiando los metadatos

Los metadatos cambian muchas veces simplemente con operar sobre el fichero para cambiar los datos. Por ejemplo, el tamaño del fichero cambia si lo haces crecer utilizando `write`, al igual que las fechas de acceso y modificación.

Aunque ya podemos vaciar un fichero utilizando `open` y hacerlo crecer utilizando `seek` y `write`, en ocasiones querremos truncar un fichero o cambiar su tamaño a un número dado de bytes. Esto puede hacerse con *truncate(2)*, que ajusta el tamaño al número indicado. Si el fichero era más grande, se tiran los bytes del final que sobran. Si el fichero era más pequeño, se hace crecer con ceros. La llamada puede utilizarse como en

```
if (truncate("myfile", 1024) < 0) {
    // truncate ha fallado...
}
```

que dejaría `myfile` con 1024 bytes exactamente. Como podrás suponer, existe *ftruncate(2)* que trunca el fichero indicado por un descriptor de fichero en lugar de por un path.

En realidad, `truncate` cambia el tamaño en los metadatos y si ello require liberar bloques en disco para los datos que ya no se usan, así lo hace.

Los permisos pueden cambiarse con la llamada *chmod(2)*, que es la que utiliza el comando *chmod(1)* que hemos usado anteriormente. Aunque esta vez sí diremos que también tienes *fchmod(2)*, en el futuro no mencionaremos más las funciones que operan con descriptores, ya estás acostumbrado a leer el manual y sabes cómo encontrarlas y usarlas.

Este programa ajusta los permisos de los ficheros que se pasan como argumento al valor indicado por el primer argumento, similar a *chmod(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <err.h>
```

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    long mode;
    int i, sts;

    if (argc < 3) {
        fprintf(stderr, "usage: %s mode file...\n", argv[0]);
        exit(1);
    }
    mode = strtol(argv[1], NULL, 8);
    sts = 0;
    for (i = 2; i < argc; i++) {
        if (chmod(argv[i], mode) < 0) {
            warn("chmod: %s", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Podemos utilizarlo como puedes ver:

```
unix$ chm 664 chm.c
unix$ ls -l chm.c
-rw-rw-r--  1 nemo  staff  452 Aug 21 11:59 chm.c
unix$
```

El propietario y el grupo a que pertenece un fichero se puede cambiar como se describe en *chown(2)*, como en:

```
if (fchown(fd, newuid, newgid) < 0) {
    // fchown ha fallado
}
```

¿Y si tienes un path en lugar de un descriptor? ¿Cómo lo harás?

Las fechas de acceso y modificación de un fichero pueden cambiarse con *utimes(2)*. Por ejemplo, este program es similar a *touch(1)*, aunque nunca crea los ficheros si no existen.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int i, sts;
    struct timeval tv[2];

    if (argc < 2) {
        fprintf(stderr, "usage: %s file...\n", argv[0]);
        exit(1);
    }
    if (gettimeofday(&tv[0], NULL) < 0) {
        err(1, "gettimeofday");
    }
    tv[1] = tv[0];
    sts = 0;
    for (i = 1; i < argc; i++) {
        // could use just NULL instead of tv.
        if (utimes(argv[i], tv) < 0) {
            warn("utimes: %s", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Aquí hemos utilizado *gettimeofday(2)* para pedir a UNIX la fecha y hora actual y cambiamos los tiempos de los ficheros justo a ese momento. El mismo efecto podría haberse conseguido utilizando "`NULL`" como argumento (que hace que `utimes` use la fecha actual).

14. Enlaces simbólicos

Recientemente hemos visto que podemos enlazar ficheros desde varios nombres. A este tipo de enlaces se los denomina **hard links** o *enlaces duros*. La ventaja que tienen es que todos los enlaces se comportan bien. Por ejemplo, borrando cualquiera de los nombres para un fichero seguimos teniendo el fichero accesible.

No obstante, no es posible hacer un enlace a un fichero que esté en otro sistema de ficheros (en otra partición, otro disco y otra máquina). Esto es obvio si piensas que el enlace simplemente crea una entrada de directorio que se refiere a un número de *i-nodo* dado, y piensas que el número de *i-nodo* sólo tiene sentido dentro de la misma partición o sistema de ficheros.

Para solucionar este problema, UNIX dispone de **enlaces simbólicos** además de enlaces duros. Un enlace simbólico es similar a un *acceso directo* en Windows. Se trata de un fichero cuyos datos son el path de otro fichero. ¡Tan sencillo como eso!. La consecuencia es que un enlace simbólico puede apuntar a cualquier fichero dado su path. Y, naturalmente, la desventaja es que si borramos el fichero original perdemos los datos del fichero y el enlace queda apuntando a un fichero que no existe.

Para crear un enlace simbólico puedes utilizar el flag `-s` de *ln(1)*. Por ejemplo:

(c) 2016 Francisco J. Ballesteros

```
unix$ echo hola >fich
unix$ ln -s fich link
unix$ ls -l fich link
-rw-r--r--  1 nemo  wheel  5 Aug 21 19:24 fich
lrwxr-xr-x  1 nemo  wheel  4 Aug 21 19:24 link -> fich
unix$ cat link
hola
unix$ cat fich
hola
unix$ rm fich
unix$ cat link
cat: link: No such file or directory
unix$ ls -l link
lrwxr-xr-x  1 nemo  wheel  4 Aug 21 19:24 link -> fich
```

Es interesante fijarse en la salida de `ls` para `link`. El primer carácter es "`l`", lo que indica que el tipo de fichero es *symbolic link*. Si recuerdas *stat(2)*, puedes utilizar código como

```
if ((st.st_mode & S_IFMT) == S_IFLNK) {
    // el fichero es un enlace simbolico
}
```

para ver si el fichero que tienes entre manos es un enlace simbólico o no.

En general, las llamadas al sistema y funciones que operan con ficheros suelen seguir los enlaces como cabe esperar. Por ejemplo, si `open` recibe como argumento el path de un fichero que es un enlace simbólico, UNIX se da cuenta de ello y procede como sigue:

- Se leen los datos del fichero
- Se interpretan como el path del que hay que hacer el `open`
- Se efectúa el `open` de dicho fichero

Esto hace que normalmente tus programas trabajen correctamente aunque encuentren enlaces simbólicos.

No obstante, hay ocasiones en que hay que prestar atención a este tipo de ficheros. Por ejemplo, si hacemos un `unlink` de un enlace simbólico tan sólo se borra el enlace y no el fichero enlazado. Aquí UNIX no sigue el enlace de forma automática. Pero es mejor así, dado que lo que cabría esperar al ejecutar

```
unix$ rm fich
```

es que se borre `fich`, sea un enlace o no. En ningún caso esperamos que se borre el fichero al que apunta `fich` si es un enlace simbólico.

Llamadas como *stat(2)* son más delicadas. Utilizar `stat` sobre un fichero que es un enlace simbólico hace que UNIX (como hace en general) atraviese el enlace automáticamente. Esto hace que en realidad obtengamos los metadatos del fichero al que apunta el enlace. Dicho de otro modo, ¡`stat` nunca te dirá que un fichero es un enlace simbólico!

Una forma de solucionar este problema es llamar a `lstat`, que funciona igual que `stat` pero cuando el fichero es un enlace devuelve los atributos del enlace en lugar de los del fichero enlazado.

Sucede lo mismo con llamadas como *chown(2)*, *chmod(2)*, etc. Estas llamadas atraviesan los enlaces simbólicos y operan sobre los ficheros enlazados, pero dispones de llamadas con igual nombre pero comenzando por "`l`" (por ej., "`lchmod`") que, cuando el fichero es un enlace simbólico, trabajan sobre el enlace en sí y no sobre el fichero enlazado.

¡El manual es tu amigo! Aunque creas que sabes usar UNIX, consulta rápidamente la página de manual de

(c) 2016 Francisco J. Ballesteros

la llamada que piensas usar. Quizá recuerdes que debieras utilizar `lstat` y no `stat` o `fstat`... ¡y ahorrarás mucho tiempo depurando bugs que podrías haber evitado!

Para crear un enlace simbólico desde C puedes utilizar *symlink(2)* que funciona igual que *link(2)*, pero crea un enlace simbólico en lugar de uno duro. Por ejemplo, este programa es similar a *ln(1)* bajo el flag `-s`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (symlink(argv[1], argv[2]) < 0) {
        err(1, "symlink %s", argv[1]);
    }
    exit(0);
}
```

15. Dispositivos

Ya conocemos diversos tipos de fichero:

- Ficheros regulares
- Directorios
- Enlaces simbólicos

Pero es hora de mencionar dos más:

- Dispositivos de tipo carácter
- Dispositivos de tipo bloque

Los dispositivos son *i-nodos* que no contienen datos en realidad. Se utilizan para representar dispositivos que pueden ser artefactos hardware (como una impresora o un disco) o pueden ser invenciones de software (como `/dev/null`). La mayoría de los dispositivos suelen estar en `/dev`.

UNIX distingue entre dispositivos de modo carácter y de modo bloque principalmente por que los primeros se espera que correspondan a streams de caracteres (el teclado, la pantalla al escribir caracteres, el ratón, etc.) y los segundos se espera que correspondan a almacenes de bloques de bytes de los que UNIX podría mantener una cache.

De hecho, los discos duros (y las particiones) suelen tener un par de ficheros de dispositivo cada uno: uno de tipo carácter y uno de tipo bloque. El primero se utiliza para formatear el disco y para leerlo saltándose la cache. El segundo se utiliza para acceder al disco beneficiándose de la cache de bloques que mantiene UNIX. Por ejemplo, presentamos los dispositivos de la primera partición del primer disco en nuestro sistema:

(c) 2016 Francisco J. Ballesteros

```
unix$ ls -l /dev/rdisk0s1
crw-r-----  1 root  operator    1,   1 Jul 13 07:30 /dev/rdisk0s1
unix$ ls -l /dev/disk0s1
brw-r-----  1 root  operator    1,   1 Jul 13 07:30 /dev/disk0s1
```

Cuando abres, lees o escribes un fichero, UNIX localiza el *i-nodo* del fichero y en función del tipo de fichero hace una cosa u otra. En el caso de los dispositivos, simplemente se llama a una función distinta (en el kernel) para cada tipo de dispositivo y se deja que ella haga el trabajo.

El código de un dispositivo suele llamarse **manejador** o **driver**. Si se trata de hardware, además de atender las interrupciones de dicho trozo de hardware y de detectar si está instalado en el sistema o no, el manejador implementará las operaciones del *i-nodo* para el tipo de dispositivo de que se trate.

En un *i-nodo* de un dispositivo tienes dos números:

- Número principal (*major number*)
- Número secundario (*minor number*)

El primero identifica un manejador de dispositivo (por ejemplo, discos duros SATA). El segundo identifica de qué dispositivo concreto se trata (por ejemplo, el primer disco SATA en el primer bus SATA).

Así pues, si un *i-nodo* es un dispositivo de bloque que identifica un disco y el número principal corresponde al driver de discos SATA, todas las llamadas `open`, `read`, ... que operen sobre ese *i-nodo* llamarán a funciones concretas del manejador de discos SATA. A dichas funciones se les dará además el *i-nodo* de que se trate y podrán ver el número secundario (ver de qué disco concreto hay que leer, escribir, etc.).

Ahora puedes entender la salida de este comando:

```
unix$ ls -l /dev/tty
crw-rw-rw-  1 root  wheel    2,   0 Aug 20 12:30 /dev/tty
```

El primer carácter es una "`c`", lo que indica *dispositivo de caracteres*. Y los números `2` y `0` corresponden al número principal y secundario.

Los dispositivos sólo son ficheros que representan a los dispositivos, no son hardware. Esto es, que tengas en `/dev` mil dispositivos para discos duros no quiere decir que tengas mil discos instalados.

Para crear un fichero de dispositivo podemos utilizar *mknod(1)*, que llama a *mknod(2)*. Pero, normalmente hay que ser *root* para poderlo hacer. !Así que vamos a convertirnos en *root* y crear uno!:

```
unix$ sudo su
Password:
unix# mknod term c 2 0
sh-3.2# echo hola >term
hola
unix# ls -l term
crw-r--r--  1 root  staff    2,   0 Aug 21 19:56 term
unix# rm term
unix# exit
unix$
```

Como verás, hemos creado un fichero llamado `term`, que es un dispositivo de modo carácter ("`c`") y tiene números `2` y `0` como *major* y *minor*. Si recuerdas el listado de `/dev/tty`, verás que estamos creando un dispositivo para el mismo artefacto. Una vez creado, podemos utilizar `term` igual que `/dev/tty`, como puedes ver por el efecto de `echo` en nuestro ejemplo.

(c) 2016 Francisco J. Ballesteros

16. Entrada/salida y buffering

El interfaz proporcionado por `open`, `close`, `read`, `write`, etc. es suficiente la mayoría de las veces. Lo que es más, en muchas ocasiones es el más adecuado. Por ejemplo, *cat(1)* debería utilizar un buffer cierto tamaño (8KiB, quizá) y usar `read` y `write` para hacer su trabajo. Es lo más simple y además lo más eficiente la mayoría de las veces.

No obstante, hay ocasiones en que deseamos leer carácter a carácter o línea a línea, o deseamos escribir poco a poco la salida del programa o escribir con cierto formato. En estos casos es mucho más adecuado utilizar la librería de C que las llamadas al sistema. Además, dicha librería suministra entrada/salida con buffering, lo que hace que el programa sea mucho mas eficiente si leemos o escribimos poco a poco.

Veamos un programa que copia un fichero en otro, pero byte a byte. Esta versión utiliza las llamadas al sistema que hemos visto.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int sfd, dfd, nr;
    char buf[1];

    if(argc != 3) {
        fprintf(stderr, "usage: %s src dst\n", argv[0]);
        exit(1);
    }


    sfd = open(argv[1], O_RDONLY);
    if(sfd < 0){
        err(1, "open: %s", argv[1]);
    }
    dfd = creat(argv[2], 0664);
    if(dfd < 0){
        close(sfd);
        err(1, "creat: %s", argv[2]);
    }
```

(c) 2016 Francisco J. Ballesteros

```
        for(;;){
            nr = read(sfd, buf, sizeof buf);
            if(nr == 0){
                break;
            }
            if(nr < 0){
                close(sfd);
                close(dfd);
                err(1, "read: %s:", argv[1]);
            }
            if(write(dfd, buf, nr) != nr){
                close(sfd);
                close(dfd);
                err(1, "write: %s:", argv[2]);
            }
        }
        close(sfd);
        if (close(dfd) < 0) {
            err(1, "close: %s:", argv[2]);
        }
        exit(0);
    }
```

Podemos crear un fichero de 10MiB utilizando el comando *dd(1)*. Basta pedirle que copie 10240 bloques de 1024 bytes desde `/dev/zero` (una fuente ilimitada de ceros) hasta el fichero en cuestión:

```
unix$ dd if=/dev/zero of=/tmp/10m bs=1024 count=10240
10240+0 records in
10240+0 records out
10485760 bytes transferred in 0.028481 secs (368166762 bytes/sec)
unix$ ls -l /tmp/10m
-rw-r--r--  1 nemo  wheel  10485760 Aug 21 22:06 /tmp/10m
```

Este comando se llama así por usarse hace tiempo para copiar *device to device*, en los tiempos en que las cintas magnéticas eran muy puntillosas respecto a qué tamaños de lectura y escritura admitían.

Ahora veamos cuánto tarda nuestro programa en copiar nuestro nuevo fichero.

```
unix$ time cp1 /tmp/10m /tmp/10mbis
real    0m16.850s
user    0m1.563s
sys    0m15.230s
unix$ ls -l /tmp/10mbis
-rw-r--r--  1 nemo  wheel  10485760 Aug 21 22:06 /tmp/1mbis
```

¡Un total de 16.8 segundos! ¡Un eón! Y para sólo 10MiB de datos. Esto quiere decir que copiar un sólo MiB tardaría aproximadamente 1.68 segundos. Muchísimo tiempo para las máquinas de hoy día.

Por cierto, el comando *time(1)* que hemos utilizado ejecuta la línea de argumentos como un comando e imprime, como has podido ver, cuánto tiempo ha empleado dicho comando en total, cuánto ejecutando código de usuario y cuánto ejecutado código del kernel en su favor. Es una herramienta indispensable para ver si cuándo haces un cambio en un programa has mejorado realmente las cosas o las has empeorado.

Veamos ahora el mismo programa, pero esta vez escrito on *stdio*, la librería estándar de C con buffering para entrada/salida. Igual que antes, vamos a copiar byte a byte.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    FILE *in, *out;
    char buf[1];
    size_t nr;

    if(argc != 3) {
        fprintf(stderr, "usage: %s src dst\n", argv[0]);
        exit(1);
    }

    in = fopen(argv[1], "r");
    if(in == NULL){
        err(1, "%s", argv[1]);
    }
    out = fopen(argv[2], "w");
    if(out == NULL){
        err(1, "%s", argv[2]);
    }


    for(;;){
        nr = fread(buf, sizeof buf, 1, in);
        if(nr == 0){
            if(ferror(in)) {
                err(1, "read");
            }
            break;
        }
        if(fwrite(buf, nr, 1, out) != nr){
            err(1, "write");
        }
    }
    fclose(in);
    fclose(out);
    exit(0);
}
```

En este caso, hemos utilizado *fopen(3)* en lugar de *open(2)* para obtener un "`FILE*`". Esta estructura representa un buffer (quizá) para operar en un descriptor de fichero que la estructura mantiene internamente. Si lees *fopen(3)* verás que hay llamadas para construir un `FILE*` a partir de un descriptor de fichero, en lugar de un path.

En realidad ya conoces este tipo de datos. Anteriormente hemos utilizado `stderr` para imprimir mensajes de error con `fprintf`. Pues bien, `stderr` es un `FILE*` para el descriptor `2`. ¿Resulta claro ahora por qué utilizamos

```
fprintf(stderr, "usage: %s [-n]\n", argv[0]);
```

para informar del uso de un programa? Igualmente tienes `stdin` para la entrada estándar y `stdout` para

(c) 2016 Francisco J. Ballesteros

la salida estándar.

La función `printf(...)` es equivalente a `fprintf(stdout,...)`. La librería *stdio* tiene funciones que comienzan por "`f`" y tienen nombres similares a funciones que ya conoces, como `fopen`, `fread`, etc. Naturalmente, operan sobre `FILE*` y no sobre descriptores de fichero: para eso están.

Pero volvamos a nuestro experimento. Vamos a ver cuánto tarda este programa en hacer el mismo trabajo:

```
unix$ time 8.bcp2 /tmp/1m /tmp/1mbis

real    0m0.912s
user    0m0.881s
sys    0m0.022s
```

Hemos tardado menos de un segundo. Bastante más rápido que antes. Y hay que decir que la máquina en que hemos hecho este experimento posee un disco SSD que en realidad es memoria de estado sólido, por lo que la diferencia de tiempos no es tanta como sería utilizando un disco duro magnético. ¡La última vez que probamos con discos magnéticos pudimos hacer el descanso de una clase mientras el primero de los programas terminaba su trabajo!

¿Cómo ha podido tardar menos esta segunda versión? La respuesta radica en que se utiliza un buffer intermedio dentro del `FILE*`. La primera vez que se llama a `fread`, *stdio* lee una cantidad razonable de bytes con una llamada a `read`. Si se estaba leyendo un sólo byte no importa, el resto ya están en el buffer esperando que los leas en el futuro. Las llamadas a procedimiento son mucho más rápidas que las llamadas al sistema, dado que éstas han de entrar y salir del kernel y dado que el kernel comprueba cada vez que lo llamas que todos tus argumentos son correctos y que todo está en orden para hacer la llamada.

En las escrituras sucede lo mismo, cuando escribes con `fwrite` (o con `fprintf`), los bytes se quedan normalmente en el buffer. Sólo cuando el buffer se llena o cierras el `FILE*` se escriben los bytes con un `write`. Eso quiere decir que aunque escribamos byte a byte, en realidad estamos haciendo muchos menos `write` y estos escriben muchos bytes a la vez.

Un detalle importante cuando se utiliza buffering, es llamar a `fclose`, dado que, si hemos escrito, los bytes pueden estar en el buffer y no haberse escrito en absoluto en el fichero. Por esta razón `stderr` no utiliza buffering en absoluto, para que los mensajes de error aparezcan inmediatamente en el terminal.

La función `fflush` vuelca el buffer de un `FILE*`, por ejemplo,

```
if (fflush(out) < 0) {
    // el flush ha fallado (fallo en write?)
}
```

escribe lo que pueda haber en el buffer de `out` en el descriptor de fichero que hay bajo `out`. Sabiendo esto... ¿Cuánto crees que tardaría el programa si haces un `fflush` en cada iteración del bucle `for`? ¿Y por qué crees tal cosa?

Hay otras muchas funciones y utilidades en *stdio(3)* que deberías conocer para programar en C. Por ejemplo, un par de funciones para leer línea a línea entre otras. Pero nosotros vamos a continuar explorando UNIX.

17. Buffering en el kernel

Hemos visto hace poco que los dispositivos de modo bloque utilizan una cache de bloques en el kernel. Esto implica que cuando escribes un fichero lo habitual es que tus datos aún no estén en el disco. Lo que es más, incluso ficheros que has creado o borrado podrían no haberse actualizado en el disco.

(c) 2016 Francisco J. Ballesteros

Dependiendo del tipo de sistema de ficheros que utilices, es posible que incluso si copias el disco en este momento (utilizando su dispositivo crudo o de modo carácter) las estructuras de datos sean incoherentes. Por ejemplo, puede que un directorio contenga una entrada de directorio con un número de *i-nodo* que todavía aparece como libre en el disco.

Una vez UNIX sincroniza su cache en disco, el sistema de ficheros en el disco será coherente, hasta que se hagan más modificaciones. La llamada al sistema *sync(2)*, y el comando *sync(1)* se utilizan precisamente para edir a UNIX que sincronice la cache en disco. Por ejemplo,

```
unix$ sync ; sync
unix$
```

es una buena forma de asegurarse de que las escrituras se han realizado, si pensamos hacer algo peligroso que podría hacer que el sistema fallase estrepitosamente.

Hemos ejecutado dos veces `sync` puesto que lo normal es que UNIX continúe sincronizando el disco tras la llamada a *sync(2)*. La segunda vez que ejecutamos este comando ha de esperar a que acabe la sincronización anterior antes de empezar.

Disponemos de otra llamada, *fsync(2)*, a la que podemos pasar un descriptor de fichero para sincronizar las escrituras de ese fichero. Lo habitual es utilizarla tras escrituras en ficheros importantes que queremos mantener coherentes en el disco aunque falle la alimentación y el sistema muera de repente antes de que sincronice su cache.

Normalmente no es preciso utilizar `sync` dado que UNIX lo hace cada 30 segundos (o cada poco tiempo). Además, cuando se detiene la operación del sistema usando *shutdown(8)* o *halt(8)* UNIX sincroniza los sistemas de ficheros.

Además, es posible que utilicemos un sistema de ficheros que presta atención al orden de las escrituras en el disco de tal forma que el estado del sistema de ficheros en el disco es siempre coherente, lo que hace menos necesario utilizar esta llamada.

Si todo falla y el sistema de ficheros se corrompe en el disco tras un apagón o un fallo del sistema, el comando *fsck(8)* ejecutará durante el siguiente arranque del sistema y tratará de dejar los sistemas de ficheros coherentes de nuevo... ¡Si es que puede! Si llega ese caso... ¡prepárate para perder ficheros! Recuerda mantener tus backups al día.

18. Ficheros proyectados en memoria.

Utilizar `read` y `write` no es la única forma de utilizar un fichero. La abstracción fichero en UNIX tiene una relación muy estrecha con la abstracción *segmento de memoria*.

Ya sabemos que un proceso tiene diversos segmentos y sabemos que el segmento de texto procede del fichero ejecutable y se *pagina en demanda*. Recordemos que un segmento en UNIX es una abstracción. Posee una dirección de memoria virtual de comienzo, un tamaño, unos permisos (leer, escribir, ejecutar) y habitualmente se pagina desde un fichero. Para BSS y otros segmentos sin inicializar (que UNIX inicializa a cero) el fichero en cuestión suele ser `/dev/zero`.

Inicialmente, el segmento será simplemente una estructura de datos y no se le asigna memoria física (salvo que sea preciso utilizar parte de la memoria en realidad como en el caso de la pila). Una vez el programa ejecuta y utiliza direcciones de memoria del segmento, el hardware provoca un fallo de página (una excepción) si no existe memoria física asignada. El manejador, el kernel, asigna la memoria física y la inicializa, poniendo la traducción adecuada de memoria virtual a física (de *página* a *marco*).

Sabes que dado que no es posible tener una traducción en la tabla que utiliza el hardware (*tabla de páginas*)

(c) 2016 Francisco J. Ballesteros

para cada dirección, las traducciones de memoria virtual a memoria física siempre traducen bloques de 4KiB de direcciones virtuales a bloques de 4KiB de direcciones físicas. A la memoria utilizada con direcciones virtuales es a lo que llamamos página y a la memoria utilizada con direcciones físicas (memoria de verdad) es a lo que llamamos marco de página. Tanto páginas como marcos deben comenzar en direcciones que sean un múltiplo del tamaño de página (normalmente 4KiB, como hemos supuesto antes en este mismo párrafo). Pero todo esto ya lo conoces de arquitectura de computadores.

Pues bien, si consideras que los ficheros (aunque procedan de disco) utilizan una cache de bloques de disco, y piensas que UNIX puede hacer que la cache para los datos del fichero sea en realidad memoria física que puede asignarse a memoria virtual... ¡Ya lo tienes!

Es posible pedir a UNIX que un segmento de memoria virtual corresponda a un fichero existente en disco. A partir de dicho momento, leer direcciones del segmento implica leer los datos del fichero (igual que se lee el código del programa conforme ejecuta). Del mismo modo, escribir direcciones del segmento implica escribir en la cache del fichero en memoria. Más adelante UNIX sincronizará el contenido del disco.

La llamada al sistema que consigue este efecto es *mmap(2)*. Tiene un número significativo de parámetros:

```
void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Para usarla bien, debes pensar en lo que hemos discutido antes. UNIX mantendrá páginas de memoria virtual y hará que correspondan a trozos del mismo tamaño en el fichero. La primera dirección virtual será `addr`, o bien UNIX elige una si `addr` es `0`. En total se proyectan `len` bytes desde el fichero a memoria. Como comprenderás, lo mejor será que `len` sea un múltiplo del tamaño de página. El fichero está identificado por el descriptor `fd` y los bytes proyectados comienzan en `offset` dentro del fichero. Los parámetros `prot` y `flags` sirven para indicar los permisos para la memoria proyectada e indicar a unix propiedades que queremos para el nuevo segmento de memoria.

Por ejemplo, algunos valores interesantes para `flags` son

- `MAP_PRIVATE`: Las escrituras hechas por el proceso no se verán ni en el fichero y en la proyecciones a memoria que otros procesos hagan. Básicamente las páginas que se modifican se copian en cuanto el proceso intenta modificarlas. A esto se le llama **copy on write** y es en realidad la técnica que utiliza UNIX para hacer creer que en un `fork` el hijo es una copia del padre. Sólo se copia lo que cambia alguno de los procesos.

- `MAP_SHARED`: Las escrituras terminan en el fichero y son compartidas con todos los demás. Esto suele ser lo habitual.

- `MAP_ANON`: En realidad queremos un nuevo segmento de memoria *anónima* (que no procede de ningún fichero).

Este programa proyecta un fichero en memoria y lo modifica.

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <err.h>
#include <string.h>

enum {KiB = 1024};
```

```
int
main(int argc, char* argv[])
{
    int fd;
    void *addr;
    char *p;

    fd = open("/tmp/afile", O_RDWR|O_CREAT, 0644);
    if (fd < 0) {
        err(1, "open: /tmp/afile");
    }
    if (ftruncate(fd, 12*KiB) < 0) {
        close(fd);
        err(1, "truncate: /tmp/afile");
    }



    addr = mmap(0, 8*KiB, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    if (addr == MAP_FAILED) {
        err(1, "mmap: /tmp/afile");
    }
    p = addr;
    strcpy(&p[2*KiB], "hi at 2k\n");
    strcpy(&p[6*KiB], "hi at 6k\n");
    if (munmap(addr, 8*KiB) < 0) {
        err(1, "munmap: /tmp/afile");
    }
    exit(0);
}
```

Observa como dejamos que UNIX determine la dirección de memoria en que ponemos el nuevo segmento (es siempre lo mejor). Además, los tamaños están elegidos para que sean múltiplos de 4KiB (que presuponemos como tamaño de página, lo que hoy día suele ser cierto en todos los sistemas). Además, cuando terminamos, llamamos a *munmap(2)* para pedir a UNIX que termine la proyección.

Si ejecutamos nuestro nuevo programa podemos ver qué tiene `/tmp/afile`:

```
unix$ map
unix$ ls -l /tmp/afile
unix$ ls -l /tmp/afile
-rw-r--r--  1 nemo  wheel  12288 Aug 27 22:07 /tmp/afile
unix$ echo '12*1024' | hoc
12288
unix$ cat /tmp/afile
hi at 2k
hi at 6k
unix$
```

Y, para ser mas precisos...

(c) 2016 Francisco J. Ballesteros

```
unix$ od -A d -c /tmp/afile
0000000   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0002048    h   i       a   t       2   k  \n  \0  \0  \0  \0  \0  \0  \0
0002064   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0006144    h   i       a   t       6   k  \n  \0  \0  \0  \0  \0  \0  \0
0006160   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0012288
```

Hemos pedido a `od` con "`-A d`" que imprima las direcciones en base 10. Como puedes ver, en el offset 2048 tenemos el string que escribió nuestro programa en su memoria. Igual sucede en el offset 6144 (6KiB) con el string escrito allí. El resto de bytes está a cero dado que nunca los hemos escrito.

Si en el futuro proyectamos el fichero de nuevo, digamos en la dirección `p`, tendremos en `&p[2*KiB]` nuestro primer string tal cual esté en el fichero. Ni que decir tiene que aunque hemos escrito y leído bytes, es memoria como todo lo demás y podríamos haber utilizado cualquier otro tipo de datos.

Con todo esto ya sabes cómo crear nuevos segmentos en tu proceso. Hace tiempo se utilizaba una llamada *sbrk(2)* para aumentar el tamaño del segmento de datos, antes de disponer de las facilidades que nos da en la actualidad la memoria virtual. La implementación de *malloc(3)* utilizaba *sbrk(2)* para pedir más memoria virtual en el segmento de datos cuando agotaba la que tenía. Hoy en día, si necesitas una cantidad ingente de memoria virtual es mejor que pidas tu propio segmento y lo uses como te plazca. De hecho, algunos sistemas UNIX crean un segmento independiente llamado *heap* para dar servicio a *malloc(3)*. Otros usan *sbrk(2)* o bien crean un segmento con suficiente tamaño de memoria virtual. En cualquier caso, es mejor dejar la implementación de `malloc` en paz y pedir lo que necesites sin molestar a la librería de C.

19. Montajes y nombres

En UNIX tienes todos los ficheros dispuestos en un único árbol como ya sabes. El directorio raíz es "`/`" y en dicho directorio están contenidos todos los ficheros a que puedes acceder. Esto es así incluso si tienes varias particiones con ficheros o varios discos.

Una vez más, estamos ante otra abstracción suministrada por el sistema, la idea de un **espacio de nombres** único que podemos adaptar para crear la ilusión de un sólo árbol, aunque tengamos múltiples árboles de ficheros. Normalmente cada árbol está guardado en una partición utilizando un formato concreto de sistema de ficheros, pero recuerda que existen árboles de ficheros implementados en software que no tienen almacenamiento en disco (como `/proc`).

La implementación es simple: UNIX dispone de una *tabla de montajes* que hace que, al recorrer paths, el kernel pueda saltar de un directorio en el árbol al directorio raíz de de otro árbol.
El efecto puedes verlo en la figura 13. En ella puedes ver que el administrador de este sistema ha utilizado dos discos (o dos particiones, a UNIX le da lo mismo) y ha hecho que en el directorio `/home` se monte el segundo disco (o la segunda partición). Tras el montaje, cuando el kernel resuelva paths y alcance "`/home`", saltará al directorio raíz del segundo disco, por lo que aparentemente tenemos paths como "`/home/nemo`", lo que es una ilusión.

A la vista de esto puedes imaginar que si `/home` contenía ficheros o directorios antes del montaje, dicho contenido resulta ahora inaccesible. Aparentemente `/home` contiene los directorios `root`, `nemo`, etc. y eso es todo lo que pueden ver los usuarios del sistema.

El comando `mount(1)` muestra los montajes en el sistema:

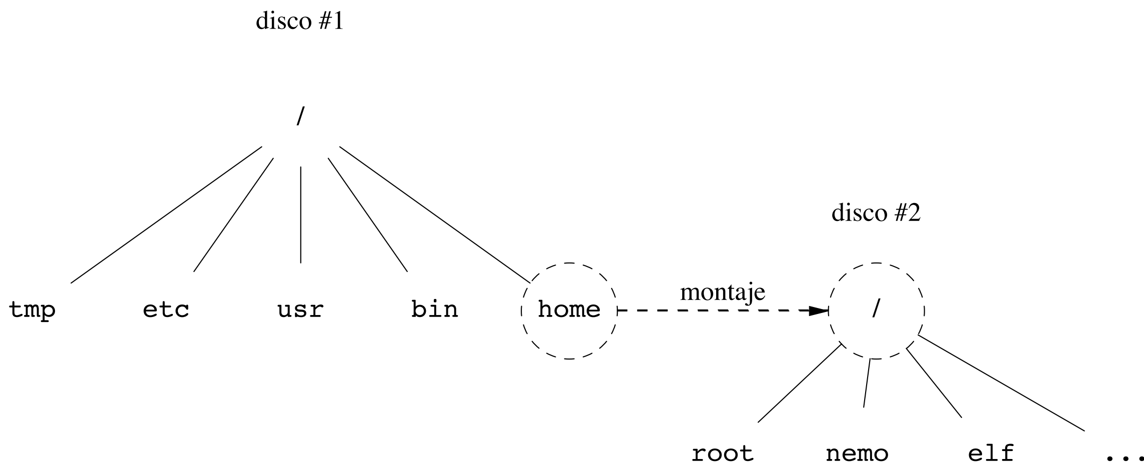

Figura 13: *Un montaje hace que UNIX salte de un directorio en el árbol al raíz de otro árbol creando el efecto de un único árbol*.

```
unix$ mount
/dev/disk1 on / (hfs, local, journaled, noatime)
devfs on /dev (devfs, local, nobrowse)
elf@fs.lsub.org:/home/dump on /dump (osxfusefs, nodev, nosuid)
elf@fs.lsub.org:/home/lsub on /zx (osxfusefs, nodev, nosuid)
elf@fs.lsub.org:/home/once on /once (osxfusefs, nodev, nosuid)
unix$
```

El resultando dependerá mucho no sólo del tipo de UNIX sino también de cómo esté administrado. En este caso vemos que hay una sóla partición en el primer disco que está montada en el directorio raíz. Por otra parte, no hay `/proc` en este sistema (OS X) y `/dev` es un sistema de ficheros sintético (igual que lo es `/proc` en otros UNIX). Puede verse también que los directorios `/dump`, `/zx` y `/once` están montados y proceden de otra máquina llamada `fs.lsub.org`.

Existe otro comando que resulta útil no sólo para ver qué tenemos montado sino también para ver cuánto espacio libre resta en cada uno de los sistemas de ficheros. hablamos de *df(1)*. Para que puedamos ver otro ejemplo, vamos a utilizar un sistema OpenBSD esta vez:

```
unix$ mount
/dev/sd2a on / type ffs (local, noatime, softdep)
unix$ df -h
Filesystem     Size    Used   Avail Capacity  Mounted on
/dev/sd2a      1.8T    239G    1.5T    14%    /
unix$
```

En este sistema hay un único sistema de ficheros montado en el raíz, procedente de la primera partición (la "`a`" en "`sd2a`") del tercer disco (el "`2`" en "`sd2a`", empezando a contar en cero). Con el flag `-h` (*human readable sizes*) `df` informa que en dicha partición de 1.8TiB estamos usando 239GiB.

El flag `-i` de `df` es muy útil e informa de cuántos *i-nodos* estamos usando en cada sistema de ficheros.

```
unix$ df -ih
Filesystem     Size    Used   Avail Capacity iused   ifree  %iused  Mounted on
/dev/sd2a      1.8T    239G    1.5T    14% 1727771 58895843     3%   /
```

(c) 2016 Francisco J. Ballesteros

Como podrás suponer, una vez hemos usado todos los *i-nodos* de que dispone un sistema de ficheros ya no es posible crear nuevos ficheros dentro del mismo.

Para montar sistemas de ficheros hay que ser *root* (a no ser que dicho usuario se ocupe de configurar el sistema para que un usuario normal pueda realizar ciertos montajes, como por ejemplo sucede con los discos USB hoy día). Un ejemplo es

```
unix$ mount -t mfs -o rw /dev/sd0b /tmp
```

que monta (en un sistema OpenBSD) en `/tmp` un sistema de ficheros en memoria virtual, respaldado por una partición de swap en disco. (Dicha partición se utiliza para mantener en ella la parte de la memoria virtual que no nos cabe en memoria física).

Para deshacer el efecto de *mount(8)* disponemos de *umount(8)*. Por ejemplo,

```
unix$ umount /tmp
```

deshace el montaje del ejemplo anterior. El sistema se negará a desmontar un sistema de ficheros que esté utilizándose. Por ejemplo, si un proceso tiene su directorio actual dentro de dicho sistema de ficheros o tiene ficheros abiertos procedentes del mismo, o está paginando su código desde un ejecutable almacenado en él. Si lo intentamos...

```
unix$ umount /
/dev/sd2a: device busy
unix$
```

Tenemos llamadas al sistema en *mount(2)* que utilizan los comandos que hemos visto, pero es muy poco probable que las necesites.

(c) 2016 Francisco J. Ballesteros

Capítulo 4: Padres e hijos

(c) 2016 Francisco J. Ballesteros

1. Ejecutando un nuevo programa

Hemos visto antes cómo es el proceso que ejecuta nuestro código. UNIX ha creado este proceso cuando se lo hemos pedido utilizando el shell y, hasta el momento, sólo hemos utilizado el shell para crear nuevos procesos.

Vamos a ver ahora cómo crear nuevos procesos y ejecutar nuevos programas pidiéndoselo a UNIX directamente. Aunque en otros sistemas tenemos llamadas similares a

```
spawn("/bin/ls");
```

para ejecutar `ls` en un nuevo proceso, ese *no* es el caso en UNIX. En su lugar, tenemos dos llamadas:

- Una sirve para crear un nuevo proceso
- Otra sirve para ejecutar un nuevo programa.

Las razones principales para esto es que podríamos querer un nuevo proceso que ejecute el mismo programa que estamos ejecutando y que podríamos querer configurar el entorno para un nuevo programa en un nuevo proceso antes de cargar dicho programa.

Antes de ver dichas llamadas detenidamente, veamos un ejemplo completo. En este programa utilizamos *fork(2)* para crear un nuevo proceso y hacemos que dicho proceso ejecute `/bin/ls` mediante una llamada a *execl(3)*:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    switch(fork()){
    case -1:
        err(1, "fork failed");
        break;
    case 0:
        execl("/bin/ls", "ls", "-l", NULL);
        err(1, "exec failed");
        break;
    default:
        printf("ls started\n");
    }
    exit(0);
}
```

El programa empieza su ejecución como cualquier otro proceso y continúa hasta la llamada a `fork`. En este punto sucede algo curioso: se crea un *clon exacto del proceso* y tanto el proceso original (llamado *proceso padre*) como el nuevo proceso (llamadado *proceso hijo*) continúan su ejecución normalmente a partir de dicha llamada. Dicho de otro modo,

- hay una única llamada a `fork` (en el proceso padre),
- pero `fork` retorna dos veces: una vez en el proceso padre y otra en el hijo.

Ambos procesos son totalmente independientes, y ejecutarán según obtengan procesador (no sabemos en qué orden).

En el proceso padre `fork` retorna un número positivo (a menos que `fork` falle, en cuyo caso retorna `-1`). Luego el padre continúa su ejecución en el `default`, imprime su mensaje y luego termina en la llamada a

(c) 2016 Francisco J. Ballesteros

`exit`.

En el proceso hijo `fork` siempre retorna `0`, con lo que el hijo entra en el `case` para `0` y ejecuta `execl`. Esta llamada *borra* por completo el contenido de la memoria del proceso hijo y carga un nuevo programa desde `/bin/ls`, saltando a la dirección de memoria en que está su punto de entrada (`main` para `ls`) y utilizando una pila que tiene argumentos `argc` y `argv` para dicha llamada *copiados* a partir de los que se han suministrado a `execl`.

Si todo va bien, `execl` *no retorna*. ¡Normal!, el programa original que hizo la llamada ya no está y no hay nadie a quién retornar. Estamos ejecutando un nuevo programa desde el comienzo, y este terminará cuando llame a `exit` (o `main` retorne y se llame a `exit`).

Si ejecutamos el programa, podemos ver una salida similar a esta:

```
unix$
ls started
total 112
-rw-r--r--  1 nemo  staff    10 Oct 21  2014 afile
-rw-r--r--  1 nemo  staff  1018 Oct 28  2014 guide
-rw-r--r--  1 nemo  staff   363 Aug 25 12:11 runls.c
-rwxr-xr-x  1 nemo  staff  8600 Aug 25 12:11 runls
unix$
```

La pregunta es... ¿tendremos siempre esta salida? Piensa que son procesos independientes, así pues ¿no podría aparecer el mensaje "`ls started`" del proceso padre en otro sitio? Piénsalo.

2. Creación de procesos

La llamada al sistema `fork(2)` crea un *clone exacto* del proceso que hace la llamada. Pero, ¿qué significa esto? Experimentemos con un nuevo programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    write(1, "one\n", 4);
    fork();
    write(1, "fork\n", 5);
    exit(0);
}
```

No hemos comprobado errores (¡mal hecho!), pero esta es la salida:

```
unix$ onefork
one
fork
fork
unix$
```

El primer `write` ejecuta y vemos `one` en la salida. Pero después, llamamos a `fork`, lo que crea otro proceso que es un clon exacto y, por tanto, están también dentro de la llamada a `fork`. Ambos procesos (padre e hijo) continúan desde ese punto y, claro, llamarán al segundo `write` de nuestro programa. Pero, naturalmente, los *dos* llaman a `write`, por lo que vemos dos veces `fork` en la salida del programa. Una pregunta

que podemos hacernos es... ¿será el primer `fork` que vemos el escrito por el padre o será el escrito por el hijo? ¿Qué opinas al respecto?

La figura 14 muestra un ejemplo de ejecución para ambos procesos en puntos diferentes del tiempo (que fluye hacia abajo en la figura).

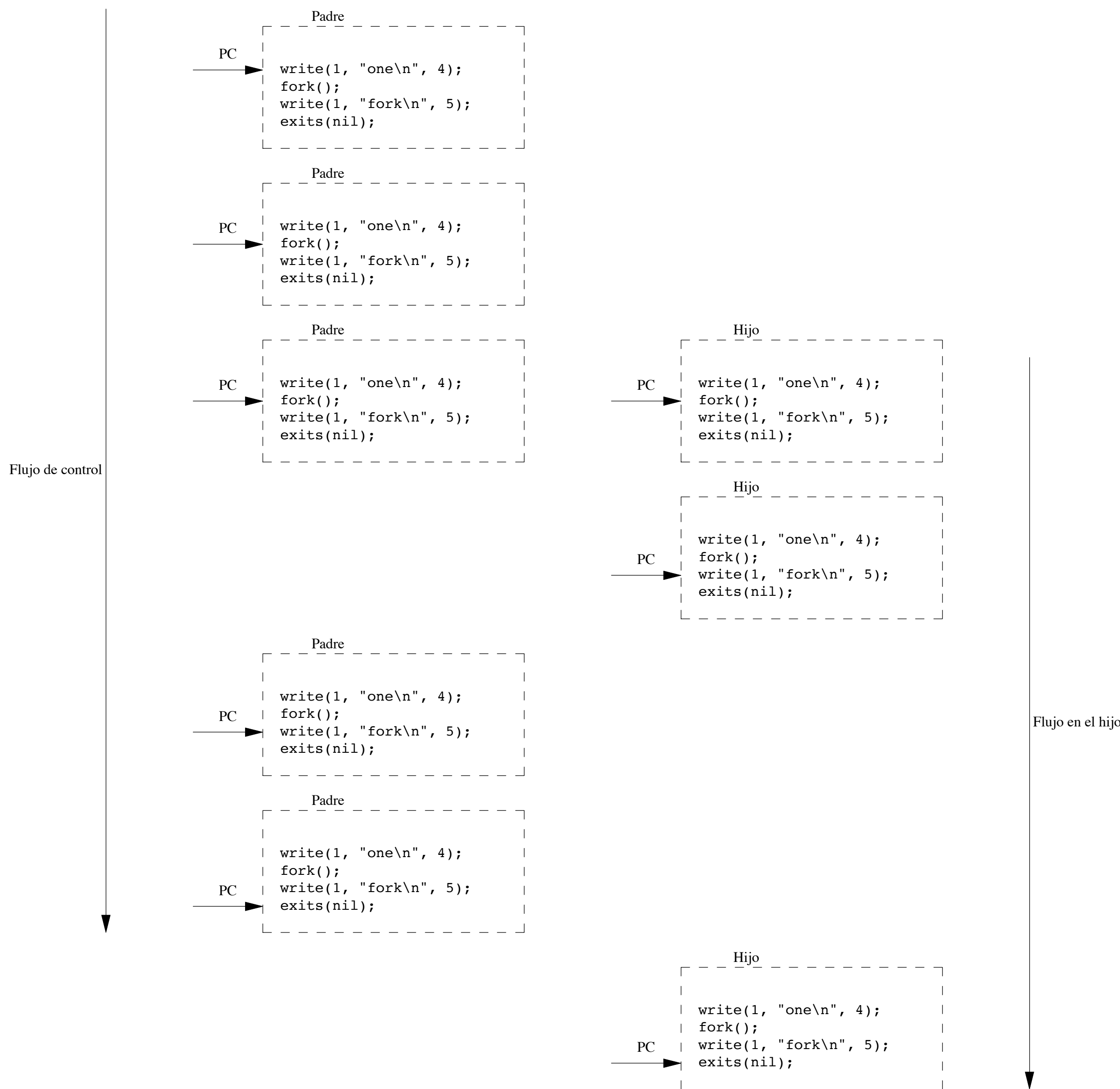

Figura 14: *La llamada a fork crea un clon del proceso original y ambos continúan su ejecución desde ese punto*.

Si seguimos la figura desde arriba hacia abajo vemos que inicialmente sólo existe el padre. Las flechas representan el contador de programa y vemos que el padre está ejecutando primero la primera llamada a `write` del programa. Después, el padre llama a `fork` y ¡aparece un nuevo proceso hijo! Cuando `fork` termina su trabajo, tanto el proceso padre como el hijo están retornando de la llamada a `fork`. Esto es lógico si piensas que el hijo es una copia exacta del padre en el punto en que llamó a `fork`, y esa copia incluye también la pila (no sólo los segmentos de código y datos).

(c) 2016 Francisco J. Ballesteros

Así pues, ambos procesos retornan del mismo modo y aparentan haber llamado a `fork` del mismo modo (aunque el hijo nunca ha hecho ninguna llamada a `fork`). Aunque UNIX hace que en el hijo `fork` retorne siempre `0`, lo que no importa en este programa. En la figura parece que el hijo ejecuta después su segundo `write` y entonces el padre continúa hasta que termina. A continuación el hijo ejecuta el código que le queda por ejecutar antes de terminar.

Naturalmente, desde el punto en que se llama a `fork`, padre e hijo pueden ejecutar en cualquier orden (o incluso de forma realmente paralela si disponemos de varios cores o CPUs en la máquina). Esto es precisamente lo que hace la abstracción *proceso*: nos permite pensar que cada proceso ejecuta independientemente del resto del mundo.

Nunca has pensado en el código del shell o el del sistema de ventanas o en ningún otro cuando has escrito un programa. Siempre has podido suponer que tu programa comienza en su programa principal y continúa según le dicte el código de forma independiente a todos los demás. Igual sucede aquí. Todo ello es gracias a la abstracción que suponen los procesos. Puedes pensar que una vez que llamamos a `fork` y se crea un proceso hijo, el hijo abandona la casa inmediatamente y continúa la vida por su cuenta.

2.1. Las variables

Dado que el proceso hijo es una copia, no comparte variables con el padre. El segmento de datos en el hijo (y la pila) son una copia de los del padre, igual que sucede con el segmento BSS y todo lo demás. Así pues, después de `fork` tu programa vive en dos procesos y cada uno tiene su propio valor para cada variable. El flujo de control (los registros y la pila) también se divide en dos (uno para cada proceso), de ahí el nombre "`fork`" ("tenedor" en inglés).

Veamos otro programa:

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int n;
    char *p;


    p = strdup("hola");
    n = 0;
    switch(fork()){
    case -1:
        err(1, "fork failed");
        break;
    case 0:
        p[0] = 'b';
        break;
    }
    n++;
    printf("pid %d: n=%d; %s at %p\n", getpid(), n, p, p);
    free(p);
    exit(0);
}
```

Intenta pensar cuál puede ser su salida y por qué antes de que lo expliquemos.

Las variables `n` y `p` están en la pila del proceso padre. La primera se inicializa a `0` y la segunda se inicializa apuntando a memoria dinámica (que está dentro de un segmento de datos, sea este el BSS o un segmento *heap* dependiendo del sistema UNIX). En dicha memoria `strdup` copia el string "`hola`".

Una vez hecho el `fork`, el proceso hijo hace que la posición a la que apunta `p` contenga `'b'`. Tras el `switch`, ambos procesos incrementan (su version de) `n`. La direcciones que que está `n` en ambos procesos coinciden (tienen el mismo valor). Pero cada proceso tiene su propia memoria virtual y su propia copia del segmento de pila. Igualmente, desde la llamada a `fork`, aunque `p` tiene el mismo valor en ambos procesos, la memoria a la que apunta `p` en el proceso hijo es distinta a la que tiene el proceso padre (¡Aunque las direcciones de memoria virtual sean las mismas!).

¿Entienes ahora por qué la salida es como sigue?

```
unix$ onefork2
pid 13083: n=1; hola at 0x7fd870c032a0
pid 13084: n=1; bola at 0x7fd870c032a0
unix$
```

Dado que *cada* proceso ha incrementado su variable `n`, ambos escriben `1` como valor de `n`. Además, los strings a que apunta `p` en cada proceso difieren, aunque las direcciones de memoria en que están en cada proceso coincidan.

Habitualmente se utiliza un `if` o `switch` justo tras la llamada a `fork` para que el código del proceso hijo haga lo que sea que tenga que hacer el hijo y el padre continúe con su trabajo. Ya dijimos que en el hijo `fork` devuelve siempre 0. En el padre `fork` devuelve el *pid* del hijo, que puede usarse para identificar qué

proceso se ha creado y para diferenciar la ejecución del padre de la del hijo en el código que escribimos. Por ejemplo,

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

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

    write(1, "first\n", 7);
    pid = fork();
    switch(pid) {
    case -1:
        err(1, "fork");
        break;
    case 0:
        printf("child pid %d\n", getpid());
        break;
    default:
        printf("parent pid %d child %d\n", getpid(), pid);
    }
    printf("last\n");
    exit(0);
}
```

escribe al ejecutar

```
unix$
first
parent pid 13172 child 13173
last
child pid 13173
last
unix$
```

¿En qué otro orden pueden salir los mensajes?

2.2. El efecto de las cachés

Vamos a reescribir ligeramente uno de los programas anteriores y ver qué sucede. Concretamente, utilizaremos *stdio* en lugar de *write(2)* para escribir mensajes. Este es el programa

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    printf("one\n");
    fork();
    printf("fork\n");
    exit(0);
}
```

Y esta es la salida

```
unix$ stdiofork
one
fork
one
fork
unix$
```

¿Qué sucede? ¿Por qué hay dos "`one` en la salida? Según entendemos lo que hace `fork`... ¿No debería salir el mensaje una única vez?

Bueno, en realidad... ¡No!. Como sabemos, `printf` escribe utilizando un `FILE*` que dispone de buffering. No tenemos garantías de que `printf` llame a `write` en cada ocasión. Tan sólo cuando el buffer se llena o la implementación de `printf` decide hacerlo se llamará a `write`.

En nuestro programa, los bytes con "`one\n`" están en el buffer de `stdout` en el momento de llamar a `fork`. En este punto, `fork` crea el proceso hijo como un *clon exacto*. Luego el hijo dispone naturalmente de los mismos segmentos de datos que el padre y el buffer de `stdout` tendrá el mismo contenido en el hijo que en el padre.

Así pues, cuando *stdio* llame a `write` para escribir el contenido del buffer, ambos mensajes aparecen en el terminal, en cada uno de los dos procesos.

3. Juegos

Este programa es curioso:

```
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    while(fork() == 0)
        ;    // catch me!
    exit(0);
}
```

El proceso padre llama a `fork` y luego muere (dado que para él `fork` devuelve `0` lo que hace que el bucle termine). No obstante, el proceso hijo continúa en el bucle y llama a `fork`. Esta vez, el hijo termina tras crear un nieto. Y así hasta el infinito. Es realmente difícil matar este programa dado que cuando estemos intentando matar al proceso, este ya habrá muerto tras encarnarse en otro.

(c) 2016 Francisco J. Ballesteros

Este otro programa es aún peor.

```
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    while(1) {
        fork();
    }
    exit(0);
}
```

Un proceso crea otro. Ambos continúan en el bucle y cada uno de ellos crea otro. Los cuatro continúan...

¡Pruébalo! (y prepárate a tener que rearrancar el sistema cuando lo hagas).

4. ¿Compartidos o no?

Cuando `fork` crea un proceso, dado que es un clon del padre, dicho proceso (hijo) tiene una copia de los descriptores de fichero del padre. Lo mismo sucede con las variables de entorno y otros recursos.

Naturalmente, sólo los descriptores de fichero se copian, ¡no los ficheros!. Piensa lo absurdo que sería (además de ser imposible) copiar el disco duro entero si un proceso tiene abierto el dispositivo del disco y hace un `fork`. Ni siquiera se copian las entradas de la tabla de ficheros abiertos (los record a que apuntan los descriptores de fichero).

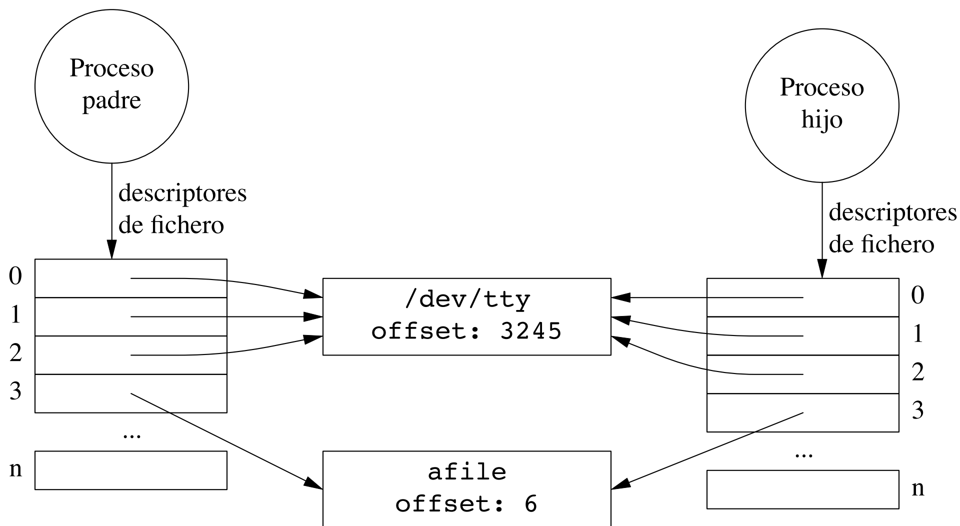

Figura 15: *Descriptores en los procesos padre e hijo tras un fork.*

La figura 15 muestra dos procesos padre e hijo tras una llamada a `fork`, incluyendo los descriptores de fichero de ambos procesos. Esta figura podría corresponder a la ejecución del siguiente programa (en el que hemos ignorado los valores devueltos por llamadas que hacemos para que el código sea más compacto, aunque hacer tal cosa es un error).

(c) 2016 Francisco J. Ballesteros

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

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

    fd = creat("afile", 0644);
    if (fd < 0) {
        err(1, "creat afile");
    }
    write(fd, "hello\n", 6);
    if(fork() == 0) {
        write(fd, "child\n", 6);
    } else {
        write(fd, "dad\n", 4);
    }
    close(fd);
    exit(0);
}
```

La efecto de ejecutar el programa podría ser este:

```
unix$ before
unix$ cat afile
hello
child
dad
unix$
```

Inicialmente, el padre tiene abierta la entrada estándar, la salida estándar y la salida de error estándar. Todas ellas van al fichero `/dev/tty`. En ese punto el padre abre el fichero `afile` (creándolo si no existe) y obtiene un nuevo descriptor (el `3` en nuestro caso, partiendo con offset `0`). Después de escribir 6 bytes en dicho fichero, el offset pasa a ser `6`.

Es ahora cuando `fork` crea el proceso hijo y vemos ambos procesos tal y como muestra la figura 15. Naturalmente, si cualquiera de los dos procesos abre un fichero en este punto, se le dará un nuevo descriptor al proceso que lo abre y el otro proceso no tendrá ningún nuevo descriptor. Los dos procesos son independientes y cada uno tiene si tabla de descriptores de fichero. Igualmente, si ambos abren el mismo fichero tras el `fork`, cada uno obtiene un descriptor que parte con el offset a `0`. Incluso si en ambos procesos el descriptor es, por ejemplo, `4`, los dos descriptores son distintos. ¿Puedes verlo?

Volviendo a nuestro programa, ambos procesos continúan y cada uno escribe su mensaje. Dado que comparten el (record que representa el) fichero abierto, comparten el offset. UNIX garantiza que writes pequeños en el mismo fichero (digamos de uno o pocos KiB) ejecutan atómicamente, o de forma indivisible sin que otros writes ejecuten durante el que UNIX está ejecutando. Así pues cuando el primer proceso haga su `write`, el offset avanzará y el segundo proceso encontrará el offset pasado el texto que ha escrito el primer proceso. Un mensaje se escribirá a continuación de otro.

Comparemos lo que ha sucedido con el efecto de ejecutar este otro programa:

(c) 2016 Francisco J. Ballesteros

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

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

    fd = creat("afile", 0644);
    if (fd < 0) {
        err(1, "creat afile");
    }
    close(fd);

    if(fork() == 0) {
        fd = open("afile", O_WRONLY);
        if (fd < 0) {
            err(1, "open afile");
        }
        write(fd, "child\n", 6);
    } else {
        fd = open("afile", O_WRONLY);
        if (fd < 0) {
            err(1, "open afile");
        }
        write(fd, "dad\n", 4);
    }
    close(fd);
    exit(0);
}
```

Esta vez...

```
unix$ after
unix$ cat afile
dad
d
unix$ xd -b -c afile
0000000  63 68 69 6c 64 0a
      0   d  a  d \n  d \n
0000006
unix$
```

¿Por qué? Simplemente porque cada proceso tiene su propio descriptor de fichero con su propio offset. Podríamos pensar que cada vez que abrimos un fichero nos dan un offset. En el programa anterior lo compartían ambos procesos, pero no esta vez. La consecuencia es que ambos procesos realizan el `write` en el offset `0`, con lo que el primero en hacer el `write` escribirá antes en el fichero. El segundo en hacerlo sobreescribirá lo que escribiese el primer proceso. En nuestro caso, como el padre parece que ha escrito después y su escritura era de menos bytes, quedan restos de la escritura del hijo a continuación del mensaje que ha escrito el padre. Ten en cuenta que aunque `write` en el hijo avanza el offset, avanza el offset en el fichero que ha abierto el hijo. Pero esta vez el padre tiene su propio offset que todavía sigue siendo `0` cuando llama a `write`.

Otra posibilidad habría sido ver esto...

(c) 2016 Francisco J. Ballesteros

```
unix$ after
unix$ cat afile
child
unix$
```

¿Es que el padre no ha escrito nada en este caso?

Si recuerdas que en *open(2)* puedes utilizar el flag `O_APPEND` comprenderás que en este programa podríamos haberlo utilizado para hacer que los writes siempre se realicen al final de los datos existentes en el fichero en lugar de en la posición que indica el offset. Pero no hemos hecho tal cosa.

4.1. Condiciones de carrera

Lo que acabamos de ver es realmente importante. Aunque el programa es el mismo, dado que hay más de un proceso involucrado, el resultado de la ejecución depende del orden en que ejecuten los distintos procesos. Concretamente, en el orden en que se ejecuten sus trozos de código (piensa que en cualquier momento UNIX puede hacer que un proceso abandone el procesador y que otro comience a ejecutar, esto es, en cualquier momento puede haber un cambio de contexto).

A esta situación se la denomina **condición de carrera**, y normalmente es un bug. No es un bug sólo si no nos importa que el resultado varíe, lo que no suele ser el caso.

Estamos adentrándonos en un mundo peligroso, llamado *programación concurrente*. La programación concurrente trata de cómo programar cuando hay múltiples procesos involucrados y dichos procesos comparten recursos. Es justo ese el caso en que pueden darse condiciones de carrera. Recuerda que decimos "concurrente" puesto que nos da exactamente igual si lo que sucede es que el sistema cambia de contexto de un proceso a otro o que los procesos ejecutan realmente en paralelo en distintos cores.

Los programas con condiciones de carrera son impredecibles y muy difíciles de depurar. Es mucho más práctico tener cuidado a la hora de programarlos y evitar que puedan suceder condiciones de carrera. Más adelante veremos algunas formas de conseguirlo.

5. Cargando un nuevo programa

Ya sabemos cómo crear un proceso. Ahora necesitamos poder cargar nuevos programas o estaremos condenados a implementar *todo* cuanto queramos ejecutar en un único programa. Naturalmente, no se hacen así las cosas.

Para cargar un nuevo programa basta con utilizar la llamada al sistema `execl`, o una de las variantes descritas en *exec(3)*. Esta llamada recibe:

- El nombre (path) de un fichero que contiene el ejecutable para el nuevo programa
- Un vector de argumentos para el programa (`argv` para su `main`)

y, opcionalmente, dependiendo de la función de *exec(3)* que utilicemos,

- Un vector de variables de entorno.

Normalmente se utiliza o bien `execl` o bien `execv`. La primera acepta el vector de argumentos como argumentos de la función, por lo que se utiliza si al programar ya sabemos cuántos argumentos queremos pasarle al nuevo programa (si se conocen en tiempo de compilación, o *de forma estática*). La segunda acepta un vector de strings para el vector de argumentos y suele utilizarse si queremos construir un vector de argumentos en tiempo de ejecución o si resulta más cómo utilizar el vector que escribir un argumento tras otro en la llamada.

Veamos un programa con `execl`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("ls", "ls", "-l", NULL);
    fprintf(stderr, "trying again\n");
    execl("/bin/ls", "ls", "-l", NULL);
    fprintf(stderr, "exec is done\n");
    exit(0);
}
```

Para que los mensajes salgan inmediatamente, el programa escribe en `stderr` (que no posee buffering) y así podemos utilizar `fprintf`. De nuevo, igual que en muchos ejemplos de los que siguen, hemos omitido las comprobaciones de error para hacer que los programas distraigan menos de la llamada con la que estamos experimentando.

Pero vamos a ejecutarlo...

```
unix$ execls
running ls
trying again
total 304
-rw-r--r--  1 nemo  staff     6 Aug 25 16:22 afile
-rwxr-xr-x  1 nemo  staff  8600 Aug 25 12:20 execls
-rw-r--r--  1 nemo  staff   363 Aug 25 12:11 execls.c
unix$
```

Claramente nuestro programa no ha leído ningún directorio ni lo ha listado. No hemos programado tal cosa. Es más, la mayoría de la salida claramente procede de ejecutar "`ls -l`". ¡Hemos ejecutado código de `ls` de igual modo que cuando ejecutamos "`ls -l`" en el shell!

Eso es exactamente lo que ha hecho la llamada a `execl`, cargar el código de `/bin/ls` en la memoria, tras lobotomizar el proceso y tirar el contenido de su memoria a la basura.

Mirando la salida más despacio, puede verse que el mensaje "`trying again`" ha salido en el terminal, pero no así el mensaje "`exec is done`". Esto quiere decir que la primera llamada a `execl` ha fallado: no ha ejecutado programa alguno y nuestro programa ha continuado ejecutando. El mero hecho de que `execl` retorne indica que ha fallado. Igual sucede con cualquiera de las variantes de *exec(3)*.

Vamos a cambiar ligeramente el programa para ver qué ha pasado:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("ls", "ls", "-l", NULL);
    fprintf(stderr, "trying again\n");
    execl("/bin/ls", "ls", "-l", NULL);
    fprintf(stderr, "exec is done\n");
    exit(0);
}
```

Y ahora sí podemos ver cuál fué el problema.

```
unix$ execls2
running ls
execls2: exec: ls: No such file or directory
trying again
total 304
...
```

No existe ningún fichero llamado "`./ls`" y naturalmente UNIX no ha podido cargar ningún programa desde dicho fichero dado que el primer argumento de `execl` (el path hacia el fichero que queremos cargar y ejecutar) es "`ls`" y no existe dicho fichero.

En la segunda llamada a `execl` resulta que hemos pedido que ejecute "`/bin/ls`" y UNIX no ha tenido problema en ejecutarlo: el fichero existe y tiene permiso de ejecución.

Inspeccionando el resto de argumentos de `execl` puede verse que la "linea de comandos" o, mejor dicho, el vector de argumentos para el nuevo programa está indicado tal cual como argumentos de la llamada. Dado que no hay magia, `execl` necesita saber dónde termina el "vector" y requiere que el último argumento sea `NULL` para marcar el fin de los argumentos.

Pero probemos a ejecutar con otro vector de argumentos:

```
#include <stdio.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("ls", "ls", "-ld" "$HOME", NULL);
    err(1, "exec failed");
}
```

Ahora esta es la salida:

```
unix$ execls3
running ls
ls: $HOME: No such file or directory
unix$
```

Como puedes ver, `execl` *no* ha fallado: no puede verse el mensaje que imprimiría la llamada a `err`, con lo que `execl` no ha retornado nunca. Esto quiere decir que ha podido hacer su trabajo. Lo que es más, `ls` ha llegado a ejecutar y ha sido el que imprime el mensaje de error quejándose de que el fichero no existe.

¡Naturalmente!, ¡Claro que no existe "`$HOME`"! Si queremos ejecutar `ls` para que liste nuestro directorio casa, habría que llamar a `getenv` para obtener el valor de la variable de entorno `HOME` y pasar dicho valor como argumento en la llamada a `execl`".

Recuerda que `execl` no es el shell. Pero... si quieres el shell, ¡Ya sabes dónde encontrarlo! Este programa

```
#include <stdio.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("/bin/sh", "sh", "-c", "ls -l $HOME", NULL);
    err(1, "exec failed");
}
```

ejecuta una línea de comandos desde C. Simplemente carga el shell como nuevo programa y utiliza su opción "`-c`" para pasarle como argumento el "comando" que queremos utilizar. Claro está, el shell sí entiende "`$HOME`" y sabe qué hacer con esa sintáxis.

Piensa siempre que no hay magia y piensa con quién estás hablando cuando escribes código: ¿C?, ¿El shell?, ...

6. Todo junto

La mayoría de las veces no vamos a llamar a `exec` (`execl`, `execv`, ...) en el proceso que ejecuta nuestro programa. Normalmente creamos un proceso y utilizamos dicho proceso para ejecutar un programa dado. Bueno... aunque *login(1)* llama directamente a `exec`. Lo hace tras preguntar un nombre de usuario y comprobar que su password es correcto, ajusta el entorno del proceso y hace un `exec` del shell del nuevo usuario (que ejecuta a nombre del usuario que ha hecho *login* en el sistema).

¡Cuidado! Aquellos que no saben utilizar UNIX algunas veces hacen un `exec` de un comando de shell por no saber utilizar el manual y no saber que existe una función en C que hace justo lo que querían hacer. No tiene sentido utilizar `fork`, `exec`, y *date(1)* para imprimir la fecha actual. Basta una línea de C si sabes leer *gettimeofday(2)* y *ctime(3)*, como hemos visto antes. Recuerda que cuando buscas código en internet no puedes saber si lo ha escrito un humano o un simio. Tu eres siempre responsable del código que incluyes en tus programas.

En cualquier caso, vamos a programar una función en C que nos permita ejecutar un programa en otro proceso dado el path de su ejecutable y su vector de argumentos. La cabecera de la función podría ser algo como

```
int run(char *path, char *argv[])
```

Haremos que devuelva `-1` si falla y `0` si ha conseguido hacer su trabajo, como suele ser costumbre.

Esta es nuestra primera versión:

(c) 2016 Francisco J. Ballesteros

```
int
run(char *path, char *argv[])
{
    switch(fork()){
    case -1:
        return -1;
    case 0:
        execv(path, argv);
        err(1, "exec %s failed", cmd);
    default:
        return 0;
    }
}
```

El proceso hijo llama a `execv` (dado que tenemos un vector, `execl` no es adecuado) y termina su ejecución si dicha llamada falla. No queremos que el hijo retorne de `run` en ningún caso. ¡Un sólo flujo de control ejecutando código en el padre es suficiente!

El proceso padre retorna tras crear el hijo, aunque esto es un problema. Lo deseable sería que `run` no termine hasta que el programa que ejecuta el proceso hijo termine. Lo que necesitamos es una forma de esperar a que un proceso hijo termine, y eso es exactamente lo que vamos a ver a continuación.

7. Esperando a un proceso hijo

La llamada al sistema *wait(2)* se utiliza para esperar a que un hijo termine. Además de esperar, la llamada retorna el valor que suministró dicho proceso en su llamada a *exit(3)* (su *exit status*). Luego podemos utilizarla tanto para esperar a que nuestro nuevo proceso termine como para ver qué tal le fué en su ejecución. Ya sabemos que el convenio en UNIX es que un estatus de salida `0` significa "todo ha ido bien" y que cualquier otro valor indica "algo ha ido mal".

Vamos a mejorar nuestra función, ahora que sabemos qué utilizar.

```
int
run(char *cmd, char *argv[])
{
    int pid, sts;

    pid = fork();
    switch(pid){
    case -1:
        return -1;
    case 0:
        execv(cmd, argv);
        err(1, "exec %s failed", cmd);
    default:
        while(wait(&sts) != pid)
            ;
        if (sts != 0) {
            return -1;
        }
        return 0;
    }
}
```

En esta versión, el proceso padre llama a `wait` hasta que el valor devuelto concuerde con el *pid* del hijo, y en ese caso el entero `sts` que ha rellenado la llamada a `wait` contiene el estatus del hijo.

(c) 2016 Francisco J. Ballesteros

El bucle en la llamada a `wait` es preciso puesto que, si nuestro proceso ha creado otros procesos antes de llamar a `wait` dentro de `run`, no tenemos garantías de que `wait` informe del proceso que nos interesa.

La llamada a `wait` espera hasta que *alguno* de los procesos hijo ha muerto y retorna con el pid y estatus de dicho hijo. Si ningún hijo ha muerto aún, `wait` se bloquea hasta que alguno muera. Y si no hay ningún proceso hijo creado... ¡Nos mereceremos lo que nos pase!

El programa que llame a `run` sólo estará interesado en si `run` ha podido hacer su trabajo o no. Por eso, si el estatus del hijo indica que el programa que ha ejecutado no ha podido hacer su trabajo, `run` retorna `-1`.

7.1. Zombies

Cuando un proceso muere en UNIX, el kernel debe guardar su estatus de salida hasta que el proceso padre hace un `wait` y el kernel puede informarle de la muerte del hijo.

¿Qué sucede si el padre nunca hace la llamada a `wait` para esperar a ese hijo? Simplemente que UNIX debe mantener en el kernel la información sobre el hijo que ha muerto. A partir de aquí, lo que ocurra dependerá el sistema concreto que utilizamos. En principio, la entrada en la tabla de procesos sigue ocupada para almacenar el estatus del hijo, por lo que tenemos un proceso (muerto) correspondiente al hijo. Pero dado que el hijo ha muerto, nunca volverá a ejecutar.

A estos procesos se los conoce como *zombies*, dado que son procesos muertos que aparecerán en la salida de *ps(1)* si el sistema que tenemos se comporta como hemos descrito. Una vez el padre llame a `wait`, UNIX podrá informarle respecto al hijo y la entrada para el hijo en la tabla de procesos quedará libre de nuevo. El zombie desaparece.

En otros sistemas el kernel mantiene en la entrada de la tabla de procesos del padre la información de los hijos que han muerto. En este caso, aunque técnicamente no tenemos un proceso zombie, el kernel sigue manteniendo recursos que no son necesarios si no vamos a llamar a `wait` en el padre.

Esta relación padre-hijo es tan importante en UNIX que cuando un proceso muere sus hijos suele adoptarlos el proceso con pid `1` (conocido como *init* habitualmente). Dicho proceso se ocupa de llamar a `wait` para que dichos procesos puedan por fin descansar en paz.

Lo importante para nosotros es que si nuestro programa crea procesos hemos de llamar a `wait` para esperarlos, o informar a UNIX del hecho de que no vamos a llamar a `wait` en ningún caso. Esto último se hace utilizando la llamada:

```
signal(SIGCHLD, SIG_IGN);
```

Aunque esta llamada no tiene nada que ver con la creación o muerte de procesos, así es como son las cosas. Más adelante veremos qué es *signal(3)* en realidad y para qué se utiliza.

8. Ejecución en background

Anteriormente hemos utilizado "`&`" en el shell, para ejecutar un comando y recuperar la línea de comandos (obtener un nuevo prompt) antes de que dicho comando termine. Como ya sabrás en este punto, para implementar "`&`" no es preciso ejecutar nada en el programa que implementa el shell. De hecho, hay que *no ejecutar* algo. Concretamente, basta con que el shell no llame a `wait` tras el `fork` que crea el proceso para el nuevo comando.

El comando *wait(1)* es un *built-in* del shell y espera hasta que los comandos que aún quedan por terminar terminen. Por ejemplo...

(c) 2016 Francisco J. Ballesteros

```
unix$ sleep 5 & echo hola ; wait
[1] 13796
hola
[1]+  Done                    sleep 5
unix$
```

y aparece "`hola`" en la salida en el acto, pero el prompt para un nuevo comando aparece 5 segundos después, cuando *wait(1)* ha terminado tras esperar que `sleep` termine.

9. Ejecutables

Para UNIX, un ejecutable es simplemente un fichero que tiene permiso de ejecución. UNIX es optimista e intentará ejecutar lo que se le pida, si es posible.

Durante la llamada al sistema `exec`, UNIX inspecciona el comienzo del fichero que ha de cargar para ejecución leyendo los primeros bytes. Dependiendo del contenido de dichos bytes pasará una cosa u otra.

9.1. Binarios

Consideremos de nuevo un ejecutable obtenido tras compilar y enlazar un "*hola mundo*" en C.

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    puts("hola mundo");
    exit(0);
}

unix$ cc -g hi.c
unix$ ls -l a.out
-rwxrwxr-x 1 elf elf 9654 Aug 26 08:38 a.out
```

El formato del fichero `a.out` dependerá mucho del tipo de UNIX que utilizamos. En general, es muy posible que sea un fichero en formato *ELF* (*Executable and Linkable Format*). No obstante, la estructura del fichero será prácticamente la misma en todos los casos:

- Una tabla al principio que indica el formato del fichero
- Una o más *secciones* con los bytes de código, datos inicializados, etc.

El comando *file(1)* en UNIX intenta determinar el tipo de fichero que tenemos entre manos. Simplemente lo lee y hace una apuesta, no hay garantías respecto a la mayoría de ficheros. Recuerda que para UNIX los ficheros son arrays de bytes y poco más.

```
unix$ file hi.c
hi.c: C source, ASCII text
unix$ file a.out
a.out: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV)
dynamically linked (uses shared libs), for GNU/Linux 2.6.24
unix$
```

Como `hi.c` contiene texto típico de fuente en C, `file` cree que contiene tal cosa (y en este caso acierta). Pero, ¿Cómo sabe que `a.out` es un ELF? Simplemente mira al comienzo del fichero y ve si hay cierta constante con cierto valor. Si la hay, se supone que es un ELF puesto que el enlazador que genera ficheros

(c) 2016 Francisco J. Ballesteros

ELF deja en esa posición ese valor. A estos valores se los llama *números mágicos* (o *magic numbers*). Simplemente sirven como una comprobación de tipos para un hombre pobre. En nuestro caso hemos utilizado Linux esta vez, como puedes ver, y el formato de los ejecutables es ELF, descrito en *elf(5)*.

Podemos utilizar *readelf(1)* para inspeccionar nuestro ejecutable. Con la opción "`-h`" podemos pedirle que vuelque los primeros bytes del fichero suponiendo que es una cabecera de un fichero en formato ELF (un record al comienzo del fichero, nada más).

```
unix$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400630
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4520 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
unix$
```

Si miramos los bytes al principio del fichero utilizando `xd` (el resto de la línea hace que sólo mostremos dos líneas de la salida de `xd`), esto es lo que vemos:

```
unix$ xd -b -c a.out | sed 2q
0000000  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
      0  7f  E  L  F 02 01 01 00 00 00 00 00 00 00 00 00
```

Como puedes ver, el fichero comienza por un número mágico que, por convenio, está presente en esa posición para todos los ficheros ELF. Así es cómo sabe UNIX que tiene un ELF entre manos. Si luego resulta que no es un ELF... ¡Mala suerte!

Pero vamos a un sistema OpenBSD y veamos qué sucede...

(c) 2016 Francisco J. Ballesteros

```
unix$ cc -g hi.c
unix$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1,
for OpenBSD, dynamically linked (uses shared libs), not stripped
unix$
unix$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0xb40
  Start of program headers:          64 (bytes into file)
  Start of section headers:          5920 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         35
  Section header string table index: 32
unix$
```

La constante (mágica) es la misma. Pero puedes ver que el resto de datos varía. Por ejemplo, en Linux el programa comenzará a ejecutar en la dirección `0x400630` (que es el punto de entrada al programa). En cambio, en OpenBSD la dirección de comienzo es `0xb40`. El enlazador en cada sistema está programado de acuerdo con los convenios del sistema para el que enlaza, y el kernel sigue dichos convenios para cargar el ejecutable.

El resto del ejecutable son *secciones*. Cada una de ellas es simplemente una serie de bytes descritos por una cabecera (otro record). Tendremos una para el código ejecutable (el texto), otra para los datos inicializados, otra para la información de depuración, y quizá algunas más.

¿Qué sucede si un fichero binario no es del formato adecuado para nuestro sistema? Pues que UNIX no puede ejecutarlo. Por ejemplo, esto sucede si copiamos el ELF de nuestro OpenBSD hacia un OSX e intentamos ejecutarlo:

```
unix$ /tmp/a.out
bash: /tmp/a.out: cannot execute binary file
unix$
```

La constante mágica lo identifica como ELF, y `exec` intentó leerlo e interpretarlo. Pero, tras mirar la cabecera este UNIX descubre que no sabe ejecutarlo y `exec` falla.

9.2. Programas interpretados

¿Y qué sucede si un fichero ejecutable no es un binario? Vamos a ver un ejemplo...

(c) 2016 Francisco J. Ballesteros

```
unix$ echo echo hola > /tmp/fich
unix$ chmod +x /tmp/fich
unix$ /tmp/fich
hola
unix$
```

Para UNIX, un fichero que tiene permiso de ejecución y no es un binario conocido es un **fichero interpretado**. UNIX denomina **script** a un fichero interpretado. Lo que hace `exec` con este tipo de fichero es ejecutar un programa que hace de *intérprete*. Ya sabes que un intérprete es simplemente un programa que interpreta otro (lo lee y ejecuta las operaciones del programa interpretado).

En el caso de UNIX, el intérprete es `/bin/sh`. Esto explica que en nuestro ejemplo, ejecutar un fichero que contiene comandos es lo mismo que ejecutar un shell y hacer que dicho shell ejecute los comandos que contiene el fichero.

Dado que existen múltiples lenguajes interpretados, es posible indicarle a UNIX qué intérprete queremos para un fichero interpretado. El convenio es que si un fichero es un *script* y comienza por "`#!`", entonces el resto de bytes hasta el primer fin de línea indica la línea de comandos que hay que utilizar para interpretar el fichero.

Por ejemplo, vamos a crear un script con este contenido

```
#!/bin/echo
ya sabemos que echo(1) no lee de stdin
```

en el fichero `ecoeco` y a ejecutarlo:

```
unix$ ecoeco
./ecoeco
unix$ ecoeco -abc hola caracola
./ecoeco -abc hola caracola
unix$
```

Cuando escribimos la línea de comandos "`ecoeco`" en el shell, éste la lee y decide hacer un `fork` y un `exec` de `./ecoeco`. El resto depende del código de `exec` en el kernel de UNIX. El shell ya ha hecho su trabajo llamando a `exec` y no tiene ni idea de si el fichero que se quiere ejecutar es un binario o un script.

Sabemos lo que hace *echo(1)*. Y que no hay magia en nada de lo que ha sucedido. El kernel de UNIX ha leído los primeros bytes de `ecoeco` y ha visto que el intérprete para dicho fichero es `/bin/echo`. Así pues, el kernel se comporta como si la llamada a `exec` fuese del estilo a

```
execl("/bin/echo", "ecoeco", "./ecoeco", NULL);
```

en el primer caso y,

```
execl("/bin/echo", "ecoeco", "./ecoeco", "-abc", "hola", "caracola", NULL);
```

en el segundo caso.

Es importante que veas que "`./ecoeco`" es `argv[1]` cuando `echo` ha ejecutado. Por eso `echo` lo ha escrito.

En resumen, en el caso de un script `exec` ejecuta el intérprete (siendo este `/bin/sh` si no se utiliza "`#!`..." y cambia el vector de argumentos para indicarle al intérprete qué fichero hay que interpretar (el que se indicó en la llamada a `exec`).

Veamos otro ejemplo para ver si esto resulta más claro ahora. Vamos a ejecutar este script

(c) 2016 Francisco J. Ballesteros

```
#!/bin/echo a b c
```

y ver lo que sucede

```
unix$ eco2 x y z
a b c ./eco2 x y z
unix$
```

Al llamar a

```
execl("./eco2", "eco2", "x", "y", "z", NULL);
```

UNIX se ha comportado como si la llamada hubiera sido

```
execl("/bin/echo", "eco2", "a", "b", "c", "eco2", "x", "y", "z", NULL);
```

Ha dejado `argv[0]` con el nombre del script y ha cambiado el resto de argumentos para incluir al principio los argumentos indicados en la línea "`#!`...". En cuanto al fichero ejecutable, ha ejecutado el indicado tras "`#!`".

¿Comprendes por qué en este caso da igual el contenido del fichero tras la línea "`#!`..."? ¡`echo` no lee ningún fichero!

Otro ejemplo más:

```
unix$ cat /tmp/catme
#!/bin/cat
uno
dos
unix$ /tmp/catme
#!/bin/cat
uno
dos
unix$
```

El comando *hoc(1)* es una calculadora. Quizá no esté instalado en tu UNIX, pero [2] tiene el fuente y explica cómo está programa. Puedes ver que `hoc` evalúa expresiones que lee de la entrada e imprime su valor:

```
unix$ hoc
2 + 2
4
^D
unix$
```

¡Vamos a crear un script!

```
unix$ cat >/tmp/exprs
#!/bin/hoc
2 + 2
3 * 5
^D
unix$
unix$ chmod +x /tmp/exprs
```

¿Comprendes por qué al ejecutarlo sucede esto?

```
unix$ /tmp/exprs
4
15
unix$
```

9.3. Scripts de shell

De ahora en adelante, puedes escribir ficheros que contienen comandos de shell para ejecutar tareas que repites múltiples veces. Por ejemplo, si estás todo el tiempo compilando y ejecutando un programa podrías hacer un script que haga tal cosa en lugar de hacerlo a mano.

Suponiendo que nuestro fichero fuente es `f.c`, podríamos crear este script

```
#!/bin/sh
cc -g f.c
./a.out
```

en el fichero `xc` y en futuro podemos ejecutar

```
unix$ xc
hola mundo
unix$
```

en lugar de compilar y ejecutar a mano el programa cada vez.

No obstante, si tenemos un error de compilación el script ejecuta el fichero `a.out` aunque no corresponda al fuente que hemos intentado compilar (sin éxito).

Podemos aprovecharnos de que el shell es en realidad un lenguaje de programación. Para el shell, los comandos pueden utilizarse como condiciones de "`if`s". Si al comando que utilizamos como condición le ha ido bien (su estatus de salida es `0`) entonces el shell considera que hay que ejecutar el cuerpo del `then`. En otro caso el shell interpreta la condición como falsa.

Este es nuestro script utilizando un "`if`" del shell:

```
#!/bin/sh
if cc f.c
then
    ./a.out
fi
```

Y ahora, si cambiamos `f.c` para que tenga un error sintáctico y no compile...

```
unix$ xc
f.c:10:20: error: expected ';' after expression
1 error generated.
unix$
```

el script no ejecuta `a.out`. Si arreglamos el error

```
unix$ xc
hola mundo
unix$
```

el comando `cc` hará un "`exit(0)`", por lo que el shell recibirá `0` cuando llame a `wait` esperando que `cc` termine. Puesto que `cc` se ha utilizado como condición en un `if`, el shell entiende que hay que considerar que la condición es cierta y ejecutará las líneas de comandos contenidas entre la línea "`then`" y la línea "`fi`".

(c) 2016 Francisco J. Ballesteros

Recuerda que el shell lee líneas de comandos, no es C:

```
unix$ if echo hola ; then date ; fi
hola
Fri Aug 26 09:59:55 CEST 2016
unix$
```

Luego si escribimos...
 unix$ if echo hola then date fi >
 ]
el shell escribe otro prompt para indicarnos que el comando `if` no está completo y necesitamos escribir más líneas. Concretamente, el shell está leyendo otra línea de comandos y esta debería ser un "`then`". Tras "`if`" todos los argumentos se ejecutarán como un comando que el shell utiliza como condición, luego ha de seguir un "comando" `then` (que es parte de la sintaxis de shell para el *if*, y no un comando en si mismo).

El shell define variables de entorno para permitir que procesemos los argumentos en scripts, y podemos utilizarlas para que nuestro script `xc` compile y ejecute cualquier fichero, así escribimos menos y no tenemos que editar el script cada vez que lo usemos con un programa distinto. Concretamente

- `$*` equivale a los argumentos del script
- `$#` contiene cuántos argumentos hay (es un string, como el valor de cualquier otra variable de entorno)
- `$0` es el nombre del script.
- `$1` es el primer argumento, `$2` el segundo, etc.

Así pues, este script `xc` compila el fichero que se indica como argumento:

```
#!/bin/sh
if cc $1
then
    ./a.out
fi
```

y lo podemos utilizar para compilar y ejecutar cualquier fuente en C

```
unix$ xc f.c
hola mundo
unix$
```

Podemos mejorarlo un poco más si hacemos que el script compruebe que de verdad recibe un argumento.

```
#!/bin/sh
if test $# -eq 0
then
    echo usage: $0 fich
    exit 1
fi
if cc $1
then
    ./a.out
fi
```

Aquí hemos utilizado el comando *test(1)* para comprobar que "`$#`" es igual al "`0`". Este comando es muy útil para evaluar condiciones en los `if` en el shell. Si no hay argumentos, el script utiliza `echo` para escribir un mensaje indicando su uso (y utilizamos "`$0`" como nombre del script).

(c) 2016 Francisco J. Ballesteros

```
unix$ xc
usage: ./xc fich
unix$
```

En otro caso el script hace su trabajo como antes.

(c) 2016 Francisco J. Ballesteros

Capítulo 5: Comunicación entre Procesos

(c) 2016 Francisco J. Ballesteros

1. Redirecciones de Entrada/Salida

Ya vimos cómo pedirle al shell que ejecute un comando haciendo que la salida estándar de dicho comando se envíe a un fichero.

```
unix$ echo hola >/tmp/fich
unix$ cat /tmp/fich
hola
unix$
```

Hemos hecho esto con diversos comandos. En todos los casos, el shell ha llamado a `fork` y, antes de llamar a `exec` para ejecutar el nuevo programa, ha hecho que la salida estándar del nuevo proceso termine en el fichero que hemos especificado (`/tmp/fich` en este ejemplo).

Pero podemos hacer lo mismo con la entrada. Por ejemplo, podemos guardar la salida de *ps(1)* en un fichero y después contar las líneas que contiene con *wc(1)*. Esto nos indica cuántos procesos estamos ejecutando:

```
unix$ ps > /tmp/procs
unix$ wc </tmp/procs
      25     144    1497
unix$
```

Aunque podríamos haber utilizado

```
unix$ wc /tmp/procs
              25     144    1497 /tmp/procs
unix$
```

Hay que decir que en este caso estaríamos ejecutando 24 procesos dado que `ps` escribe una línea de cabecera indicando qué contiene cada columna de su salida.

Aunque la salida de `wc` es similar en ambos casos, en el primer caso `wc` no ha recibido argumentos y se limita a leer su entrada estándar y contar líneas, palabras y caracteres. En el segundo caso hemos utilizado `/tmp/procs` como argumento, por lo que `wc` abre dicho fichero y cuenta sus líneas, palabras y caracteres. De ahí que en el segundo caso `wc` muestre el nombre de fichero tras la cuenta.

Los caracteres "`<`" y "`>`" son sintaxis de shell y se utilizan para indicar **redirecciones de entrada/salida**. Su utilidad es hacer que la entrada o la salida estándar de un comando (del proceso que lo ejecuta) se redirija a un fichero. El comando se limita a leer del descriptor `0` y escribir en el `1`, como cabe esperar.

Podemos también redirigir cualquier otro descriptor, por ejemplo la salida de error estándar.

```
unix$ if ls /blah 2>/dev/null >/dev/null
> then
>     echo /blah existe
> else
>     echo /blah no existe
> fi
/blah no existe
unix$
```

En este caso hemos enviado la salida de error estándar ("`2`") al fichero `/dev/null` y la salida estándar también a `/dev/null`. Dicho fichero es un dispositivo que ignora los writes (pretendiendo que se han hecho correctamente) y que devuelve *eof* cada vez que se lee del mismo. Así pues, hemos utilizado `ls` sólo por su *exit status*, para ver si un fichero existe o no.

Habría sido más apropiado utilizar

```
unix$ if test -e /blah
...
```

para ver si `/blah` existe, o

```
unix$ if test -f /blah
```

para ver si existe y es un fichero regular, o

```
unix$ if test -d /blah
...
```

para ver si existe y es un directorio. Deberías leer *test(1)* para echar un vistazo a todas las condiciones que permite comprobar. La utilidad de este comando es tan sólo llamar a `exit(0)` o `exit(1)` dependiendo de si la condición que se le pide comprobar es cierta o no.

Volviendo a las redirecciones, también podemos indicar que un descriptor se redirija al sitio al que se refiere otro descriptor, como en

```
unix$ echo houston we have a problem 1>&2
houston we have a problem
unix$
```

que escribe su mensaje en la salida estándar (esto lo hace `echo`) pero haciendo que la salida estándar se diriga al sitio (al fichero) al que se refiere la salida de error estándar (esto lo hace el shell con la redirección).

Podemos comprobar que este es el caso enviando además la salida estándar a `/dev/null`. Si el mensaje sigue saliendo, es que se escribe en la salida de error.

```
unix$ ( echo hola 1>&2 )>/tmp/a
hola
unix$ cat /tmp/a
unix$ ( echo hola 2>&1 )>/tmp/a
unix$ cat /tmp/a
hola
unix$
```

Los paréntesis son sintaxis de shell para agrupar comandos y aplicar una redirección si queremos a un conjunto de comandos. Los hemos utilizado para que resulte obvio lo que está pasando.

Por ejemplo, nuestro script para compilar y ejecutar debiera ser

```
#!/bin/sh
if test $# -eq 0
then
    echo usage: $0 fich 1>&2
    exit 1
fi
if cc $1
then
    ./a.out
fi
```

dado que el mensaje de error "`usage`..." debería escribirse en la salida de error estándar.

Veamos cómo hacer una redirección en C.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd;
    int sts;

    switch(fork()){
    case -1:
        err(1, "fork failed");
        break;
    case 0:
        fd = open("iredir.c", O_RDONLY);
        if (fd < 0) {
            err(1, "open: iredir.c");
        }
        dup2(fd, 0);
        close(fd);
        execl("/bin/cat", "cat", NULL);
        err(1, "exec failed");
        break;
    default:
        wait(&sts);
    }
    exit(0);
}
```

En este programa, tras llamar a `fork`, el proceso hijo hace algunos ajustes y después llama a `exec` para cargar y ejecutar el programa `cat`. Y sabemos que cuando `cat` ejecuta sin argumentos lee su entrada y la escribe en la salida. Si ejecutamos este programa vemos algo como esto

```
unix$ iredir
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
...
    exit(0);
}
unix$
```

¡El programa escribe el contenido de `iredir.c`!

Como puedes imaginar, `cat` sigue siendo el mismo binario de siempre. No tiene código (ni argumentos) que le indiquen que ha de leer `iredir.c`. Se limita a leer del descriptor `0` y escribir en el `1`. Eso si, antes de llamar a `exec`, nuestro programa ha ejecutado

(c) 2016 Francisco J. Ballesteros

```
fd = open("iredir.c", O_RDONLY);
if (fd < 0) {
    err(1, "open: iredir.c");
}
dup2(fd, 0);
close(fd);
```

La parte interesante es la llamada a *dup2(2)*. Primero, hemos abierto `iredir.c` para leer, lo que nos ha dado un descriptor de fichero que vamos a suponer que es `3`. A continuación cuando el programa efectúa la llamada

```
dup2(3, 0);
```

hace que UNIX deje como descriptor `0` lo mismo que tiene el descriptor `3`. Recuerda que un descriptor es un índice en la tabla de descriptores del proceso, que apunta a (el record que representa) un fichero abierto. Tras la llamada, el descriptor `0` corresponde a `iredir.c` (para leer y por el momento con offset `0`). Dado que no necesitamos el descriptor `3` para nada más, el programa lo cierra. Puedes ver en la figura 16 cómo están los descriptores antes y después de *duplicar* el descriptor `3` en el `0`.

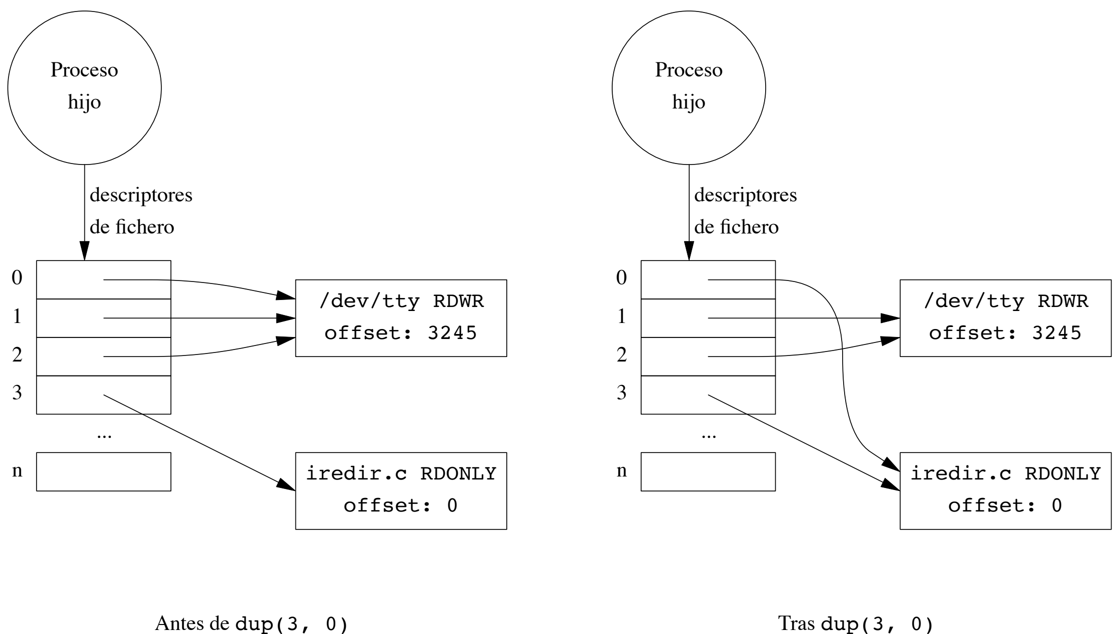

Figura 16: *Procesos antes y después de duplicar el descriptor 3 en el 0.*

Cuando ejecute `cat`, su código leerá de `0` que esta vez consigue leer de `iredir.c`. Eso es todo.

Es preciso seguir los convenios que tenemos en UNIX. Por ejemplo, si redirigimos la salida estándar utilizando un descriptor que hemos abierto en modo lectura, las escrituras fallarán. Vamos a cambiar el código de nuestro programa para que ejecute

(c) 2016 Francisco J. Ballesteros

```
fd = open("iredir.c", O_RDONLY);
if (fd < 0) {
    err(1, "open: iredir.c");
}
dup2(fd, 0);
dup2(fd, 1);
close(fd);
```

En lugar de lo que hacía antes, y podríamos ver esto

```
unix$ iredir2
cat: stdout: Bad file descriptor
unix$
```

¡`cat` no puede escribir en la salida! ¿Puedes ver por qué?

Una redirección de salida hace que el shell llame a `creat` para crear el fichero al que hay que enviar la salida. Una de entrada hace que el shell llame a `open` para leer del fichero. Además, es posible utilizar ">>" para pedirle al shell que envíe la salida a un fichero en modo *append*. En este caso, el shell hará el `open` del fichero en cuestión utilizando el flag `O_APPEND` de `open`, que indica a UNIX que se desea efectuar las escrituras al final del fichero.

Pero es preciso tener cuidado cuando combinamos redirecciones. Ya sabemos lo que hace `creat`. Por ejemplo, supongamos que queremos dejar un fichero de texto con su contenido en mayúsculas. El comando *tr(1)* sabe *traducir* unos caracteres por otros. En particular,

```
tr a-z A-Z
```

cambia los caracteres en el rango "`a-z`" por los del rango "`A-Z`", lo que efectivamente pasa texto a mayúsculas. Por ejemplo,

```
unix$ echo hola >fich
unix$ cat fich
hola
unix$ tr a-z A-Z <fich
HOLA
unix$
```

¡Vamos a utilizar un comando para pasar `fich` a mayúsculas!

```
unix$ tr a-z A-Z <fich >fich
unix$ cat fich
unix$
```

¿Qué ha pasado? !Hemos perdido el contenido de `fich`!

¡Naturalmente! El shell lee la línea y la ejecuta. En este caso sabemos que ejecutará `tr` en un nuevo proceso y que hará dos redirecciones antes de ejecutarlo (tras el `fork`):

- la entrada se redirige a `fich` (abierto para leer)
- la salida se redirige a `fich` (usando `creat`).

En cuanto el shell ha llamado a `creat`, ¡perdemos el contenido de `fich`! Deberíamos haber utilizado en este caso algo como

(c) 2016 Francisco J. Ballesteros

```
unix$ tr a-z A-Z <fich >/tmp/tempfich
unix$ mv /tmp/tempfich fich
unix$ cat fich
HOLA
unix$
```

Ahora que conocemos *dup(2)* podemos entender que "`2>&1`" es en realidad un `dup`. ¿Cuál sería? ¿Será

```
dup2(2, 1);
```

o

```
dup2(1, 2);
```

será lo que haga?

Cuando veas una línea de comandos como

```
unix$ cmd >/foo 2>&1
```

piensa en lo que hace cada redirección y en que las del tipo "`2>&1`" son llamadas a `dup2`. Y recuerda que el orden en que hacen dichas redirecciones importa cuando hay llamadas a `dup2` de por medio.

2. Pipelines

Hace tiempo, UNIX disponía de las redirecciones que hemos visto y los usuarios combinaban programas existentes para procesar ficheros. Pero era habitual procesar un fichero con un comando y luego procesar la salida que éste dejaba con otro comando, y así sucesivamente. Por ejemplo, si queremos contar el número de veces que aparece la palabra "failed" en un fichero, sin tener en cuenta si está en mayúsculas o no, podríamos convertir nuestro fichero a minúsculas, quedarnos con las líneas que contienen "failed" y contarlas:

```
unix$ tr A-Z a-z fich >/tmp/out1
unix$ grep failed <tmp/out1 >/tmp/out2
unix$ wc -l </tmp/out2
1
unix$
```

Hemos utilizado el comando *grep(1)* que escribe aquellas líneas que contienen la expresión que hemos indicado como argumento. Más adelante volveremos a usarlo.

Pero a Doug McIlroy se le ocurrió que deberían poderse usar los programas para recolectar datos, como en un jardín, haciendo que los datos pasen de un programa a otro. En ese momento introdujeron en UNIX un nuevo artefacto, el **pipe** o *tubería*, y cambiaron todos los programas para que utilizasen la entrada estándar si no recibían un nombre de fichero como argumento.

El resultado es que podemos escribir

```
unix$ cat fich | tr A-Z a-z | grep failed | wc -l
1
unix$
```

en lugar de toda la secuencia anterior. Cada "`|`" que hemos utilizado es una *tubería* (un pipe) que hace que los bytes que escribe el comando anterior en su salida sean la entrada del comando siguiente. Es como si conectásemos todos estos comandos en una tubería. Lo que vemos en la salida es la salida del último comando (y claro, todo lo que escriban en sus salidas de error estándar).

Por cierto, que si hubiésemos leído *grep(1)*, podríamos haber descubierto el flag `-i` que hace que `grep` ignore la capitalización, consiguiendo el mismo efecto con

```
unix$ grep -i failed fich | wc -l
1
unix$
```

que con el comando anterior. ¡El manual es tu amigo!

La figura 17 muestra cómo los procesos en esta última línea de comandos quedan interconectados por un pipe.

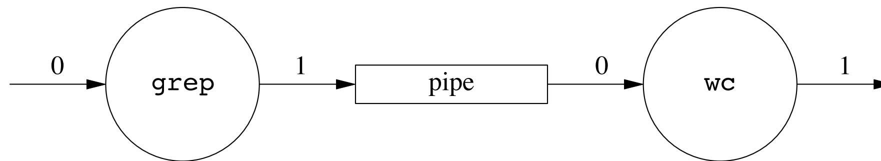

Figura 17: *Utilizando un pipe para enviar la salida de* `grep` *a la entrada de* `wc`.

En la figura hemos representado los descriptores como flechas y utilizado números para indicar de qué descriptor se trata en cada caso.

Debes pensar en el pipe como en un fichero peculiar que tiene dos extremos, uno para leer y otro para escribir. O puedes pensar que los bytes son agua y el pipe es una tubería. Los pipes ni leen ni escriben. Son los procesos los que leen y escriben bytes. Otra cosa es dónde van esos bytes o de dónde proceden.

Para crear un pipe puedes utilizar código como este

```
int fd[2];
if (pipe(fd) < 0) {
    // pipe ha fallado
}
```

que rellena el array `fd` con dos descriptores de fichero. En `fd[0]` tienes el descriptor del que hay que leer para leer de la tubería y en `fd[1]` tienes el que puedes utilizar para escribir en la tubería. Una buena forma de recordarlo es pensar que `0` era la entrada y `1` la salida.

3. Juegos con pipes

Antes de programar algo que consiga el efecto de la línea de comandos que hemos visto, vamos a jugar un poco con los pipes para ver si conseguimos entenderlos correctamente. Aquí tenemos un primer programa que utiliza `pipe`.

(c) 2016 Francisco J. Ballesteros

```
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd[2], nr;
    char buf[1024];

    if (pipe(fd) < 0) {
        err(1, "pipe failed");
    }
    write(fd[1], "Hello!\n", 7);
    nr = read(fd[0], buf, sizeof(buf));
    write(1, buf, nr);
    exit(0);
}
```

Cuando lo ejecutamos, sucede lo siguiente:

```
unix$ pipe1
Hello!
unix$
```

Tras llamar a `pipe`, el programa escribe 7 bytes en el pipe y luego lee del pipe. Como puedes ver, ha leído lo mismo que ha escrito. Eso quiere decir que lo que escribes en un pipe es lo que se lee del mismo.

Cambiemos ahora el programa para que haga dos writes en el pipe usando

```
if (pipe(fd) < 0) {
    err(1, "pipe failed");
}
write(fd[1], "Hello!\n", 7);
write(fd[1], "Hello!\n", 7);
nr = read(fd[0], buf, sizeof(buf));
write(1, buf, nr);
```

¿Qué escribirá ahora? Si lo ejecutamos podremos verlo:

```
unix$ pipe2
Hello!
Hello!
unix$
```

¡Un sólo `read` ha leído lo que escribimos con los dos `write`s! Dicho de otro modo, los pipes de UNIX (en general) no delimitan mensajes. O, no preservan los límites de los writes. Sucede igual que en conexiones de red. Una vez los bytes están en el pipe da igual si se escribieron en un único `write` o en varios. Cuando un `read` lea del pipe, leerá lo que pueda.

Vamos a intentar escribir todo lo que podamos dentro de un pipe en este otro programa:

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd[2], nw;
    char buf[1024];

    if (pipe(fd) < 0)
        err(1, "fork failed");
    for(;;){
        nw = write(fd[1], buf, sizeof buf);
        fprintf(stderr, "wrote %d bytes\n", nw);
    }
    exit(0);
}
```

Cuando lo ejecutamos

```
unix$ fillpipe
wrote 1024 bytes
wrote 1024 bytes
...
wrote 1024 bytes
```

vemos 64 mensajes impresos y el programa no termina. El programa está dentro de una llamada a `write`, intentando escribir más en el pipe, ¡pero no puede!

Los pipes tienen algo de buffer (son sólo un buffer en el kernel que tiene asociados dos descriptores). Cuando escribimos en un pipe los bytes se copian al buffer del pipe. Cuando leemos de un pipe los bytes proceden de dicho buffer. Pero si llenamos el pipe, UNIX detiene al proceso que intenta escribir hasta que se lea algo del pipe y vuelva a existir espacio libre en el buffer del pipe. Como puedes ver, en nuestro sistema UNIX resulta que los pipes pueden almacenar 64KiB, pero no más.

Y aún nos falta por ver un último efecto curioso que puede producirse si escribimos en un pipe. Observa el siguiente programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd[2];

    if (pipe(fd) < 0) {
        err(1, "pipe failed");
    }
    close(fd[0]);
    fprintf(stderr, "before\n");
    write(fd[1], "Hello!\n", 7);
    fprintf(stderr, "after\n");
    exit(0);
}
```

Si esta vez lo ejecutamos...

```
unix$ closedpipe
before
15131: signal: sys: write on closed pipe
unix$
```

UNIX mata el proceso en cuanto intenta escribir. Veremos cómo cambiar este comportamiento, pero es el comportamiento normal en UNIX cuando escribimos en un pipe del que nadie puede leer.

Piensa en una línea de comandos en que utilizas un pipeline y el último comando termina pronto. Por ejemplo, escribiendo los dos primeros strings que contiene el disco duro y que son imprimibles:

```
unix# cat /dev/rdisk0s1 | strings | sed 2q
BSD  4.4
gEFI        FAT32
unix#
```

¿Querrías que `cat` continuase leyendo *todo* el disco una vez has encontrado lo que buscas? (El comando *strings(1)* escribe en la salida los bytes de la entrada que corresponden a strings imprimibles, ignorando el resto de lo que lee).

Una vez `sed` imprime las dos primeras líneas que lee, termina. Esto tiene como efecto que el segundo pipe deja de tener descriptores abiertos para leer del mismo. El efecto es que cuando `strings` intenta escribir tras la muerte de `sed`, UNIX mata a `strings`. A su vez, esto hace que el primer pipe deje de tener abiertos descriptores para leer del mismo. En ese momento, si `cat` intenta escribir, UNIX lo mata y termina la ejecución de nuestra línea de comandos.

Nos falta por ver qué sucede si leemos repetidamente de un pipe. Podemos modificar uno de los programas anteriores para verlo de forma controlada:

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd[2], nr;
    char buf[5];

    if (pipe(fd) < 0) {
        err(1, "pipe failed");
    }
    write(fd[1], "Hello!\n", 7);
    close(fd[1]);
    do {
        nr = read(fd[0], buf, sizeof(buf)-1);
        if (nr < 0) {
            err(1, "pipe read failed");
        }
        buf[nr] = 0;
        printf("got %d bytes '%s'\n", nr, buf);
    } while(nr > 0);
    exit(0);
}
```

¡Vamos a ejecutarlo!

```
unix$ piperd
got 4 bytes 'Hell'
got 3 bytes 'o!
'
got 0 bytes ''
unix$
```

El primer `read` obtiene 4 bytes (que es cuanto le dejamos leer por el tamaño del buffer). Observa que terminamos los bytes que leemos con un byte a cero para que C lo pueda entender como un string.

El segundo `read` consigue leer los 3 bytes restantes que habíamos escrito. Pero el tercer `read` recibe una indicación de *EOF* (0 bytes leídos). Esto es natural si pensamos que nadie puede escribir en el pipe (hemos cerrado el descriptor para escribir en el pipe y nadie más lo tiene) y que hemos vaciado ya el buffer del pipe.

Así pues, cuando ningún proceso tiene abierto un descriptor para poder escribir en un pipe y su buffer está vacío, `read` siempre devuelve una indicación de EOF. Es importante por esto que cierres todos los descriptores en cuanto dejen de ser útiles. En este ejemplo ves que si hubiésemos dejado abierto el descriptor de `fd[1]` el programa nunca terminaría.

4. Pipeto

Vamos a utilizar ahora los pipes para hacer un par de funciones útiles. La primera nos dejará (en un programa en C) ejecutar un comando externo de tal forma que podamos escribir cosas en su entrada estándar. Hay mucho usos para esta función. Uno de ellos es enviar correo electrónico.

El comando *mail(1)* es capaz de leer un mensaje de correo (texto) de su entrada y enviarlo. Podemos

utilizar el flag `-s` para indicar un *subject* y suministrar como argumento la dirección de *email* a que queremos enviar el mensaje. Por ejemplo, si tenemos las notas de una asignatura en un fichero llamado "NOTAS" y en cada línea tenemos la dirección de email y las notas de un alumno, podríamos ejecutar

```
unix$ EMAIL=geek@geekland.com
unix$ grep $EMAIL NOTAS | mail -s 'tus notas' $EMAIL
unix$
```

para enviar las notas al alumno con su email en `$EMAIL`.

Estaría bien poder hacer lo mismo desde C y poder programar algo como

```
fd = pipeto("mail -s 'tus notas' geek@geekland.com");
if (fd < 0) {
    // pipeto failed
    return -1;
}
nw = write(fd, mailtext, strlen(mailtext));
...
close(fd);
```

para enviar el mensaje desde un programa en C. En este caso queremos que la función `pipeto` nos devuelva un descriptor que podamos utilizar para escribir algo que llegue a la entrada estándar del comando que ejecuta `pipeto`.

Esta es la función:

```
int
pipeto(char* cmd)
{
    int fd[2];

    pipe(fd);
    switch(fork()){
    case -1:
        return -1;
    case 0:
        close(fd[1]);
        dup2(fd[0], 0);
        close(fd[0]);
        execl("/bin/sh", "sh", "-c", cmd, NULL);
        err(1, "execl");
    default:
        close(fd[0]);
        return fd[1];
    }
}
```

Como puedes ver, llamamos a `pipe` antes de hacer el `fork`. Esto hace que tras el `fork` tanto el padre como el hijo tengan los descriptores para leer y escribir en el pipe. El padre cierra el descriptor por el que se lee del pipe (no lee nunca del pipe) y retorna el descriptor que se usa para escribir. En cambio, el hijo cierra el descriptor por el que se escribe en el pipe y a continuación ejecuta

```
dup2(fd[0], 0);
close(fd[0]);
execl("/bin/sh", "sh", "-c", cmd, NULL);
```

(c) 2016 Francisco J. Ballesteros

para ejecutar `cmd` como un comando en un shell cuya entrada estándar procede del pipe.

El efecto de ejecutar, por ejemplo,

```
fd = pipeto("grep foo");
```

puede verse en la figura 18.

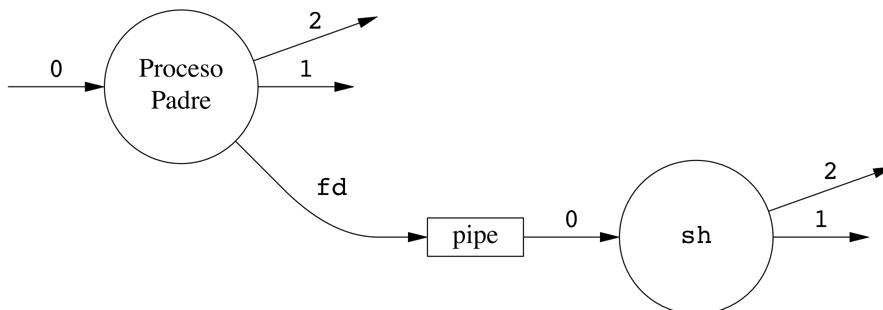

Figura 18: *Descriptores tras la llamada a pipeto mientras ejecutan ambos procesos.*

Para poder ejecutar comandos de shell la función ejecuta un shell al que se le indica como argumento el comando que queremos ejecutar. Si se desea ejecutar un sólo comando o no se requiere poder utilizar sintaxis de shell podríamos ejecutar directamente el programa que deseemos.

Un detalle importante es que si no hubiésemos cerrado en el hijo el descriptor por el que se escribe en el pipe, el comando nunca terminaría si lee la entrada estándar hasta EOF. ¿Puedes ver por qué?

Otro detalle curioso es que redirigimos la entrada del proceso hijo (para que lea del pipe) pero *no* redirigimos la salida del padre para escribir en el pipe. ¿Qué te parece esto?

¡Naturalmente!, el hijo hará un `exec` y el programa que ejecutemos *no sabe* que ha de leer de ningún pipe. Simplemente va a leer de su entrada estándar. Por ello hemos de conseguir que el descriptor `0` en dicho proceso sea el extremo del pipe por el que se lee del mismo. Pero el código del padre es harina de otro costal. El padre *sabe* que tiene que escribir en el descriptor que devuelve `pipeto`. Así pues, ¿por qué habríamos de redirigir nada para escribir en el pipe?

Además, una vez rediriges la salida estándar, has perdido el valor anterior del descriptor y no puedes volver a recuperar la salida estándar anterior. Ni siquieres podrías abriendo `/dev/tty`, dado que quizá tu salida estándar no era `/dev/tty`.

5. Pipefroms

Otra función de utilidad realiza el trabajo inverso, permite leer la salida de un comando externo en un programa escrito en C. Este podría ser el código de ficha función.

(c) 2016 Francisco J. Ballesteros

```
int
pipefrom(char* cmd)
{
    int fd[2];

    if (pipe(fd) < 0) {
        return -1;
    }
    switch(fork()){
    case -1:
        return -1;
    case 0:
        close(fd[0]);
        dup2(fd[1], 1);
        close(fd[1]);
        execl("/bin/sh", "sh", "-c", cmd, NULL);
        err(1, "execl");
    default:
        close(fd[1]);
        return fd[0];
    }
}
```

En este caso redirigimos la salida estándar del nuevo proceso al pipe (en el proceso hijo) y retornamos el descriptor para leer del pipe.

Podemos utilizar esta función en un programa para leer la salida de un comando. Por ejemplo, este programa ejecuta el comando `who` que muestra qué usuarios están utilizando el sistema y lee su salida. En nuestro caso nos limitamos a escribir toda la salida del comando en nuestra salida estándar. En un caso real podríamos procesar la salida de `who` para hacer con ella algo más interesante.

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
pipefrom(char* cmd)
{
    // como se muestra arriba
}

static int
readall(int fd, char buf[], int nbuf)
{
    int tot, nr;

    for (tot = 0; tot < nbuf; tot += nr) {
        nr = read(fd, buf+tot, nbuf-tot);
        if (nr < 0) {
            return -1;
        }
        if (nr == 0) {
            break;
        }
    }
    return tot;
}

int
main(int argc, char* argv[])
{
    int    fd, nr;
    char    buf[1024];

    fd = pipefrom("who");
    if (fd < 0)
        err(1, "pipefrom");
    nr = readall(fd, buf, sizeof buf - 1);
    close(fd);
    if (nr < 0) {
        err(1, "read");
    }
    buf[nr] = 0;
    printf("command output:\n%s--\n", buf);
    exit(0);
}
```

El programa incluye una función auxiliar, `readall`, que lee todo el contenido desde el descriptor indicado dejándolo en un buffer. Esta podría ser una ejecuciæon del programa:

```
unix$ pwho
nemo     console  Jul 13 07:30
nemo     ttys000  Jul 13 07:31
nemo     ttys001  Aug 18 15:59
nemo     ttys002  Aug 20 18:48
unix$
```

6. Sustitución de comandos

El shell incluye sintaxis que realiza un trabajo similar al código que acabamos de ver. Se trata de la la llamada *sustitución de comandos* o *interpolación de salida* de comandos. La idea es utilizar un comando *dentro* de una línea de comandos para que dicho comando escriba el texto que debiéramos escribir nosotros en caso contrario.

Por ejemplo, podríamos querer guardar en una variable de entorno la fecha actual. Para ello deberíamos ejecutar `date` y luego escribir una línea de comandos que asigne a una variable de entorno lo que `date` ha escrito. Pero, por un lado, esto no es práctico y, por otro, si queremos ejecutar estos comandos como parte de un script, no queremos que un humano tenga que editar el script cada vez que lo ejecutamos.

Por ejemplo, tal vez queremos hacer un script llamado `mkvers` que genere un fichero fuente en C que declare un array que defina la versión del programa con la fecha actual.

Veamos cómo hacerlo. Queremos tener un fichero que contenga, por ejemplo,

```
char vers[] = "vers: Fri Aug 26 17:05:14 CEST 2016";
```

para utilizarlo junto con otros ficheros al construir nuestro ejecutable. El plan es que cada vez que hagamos un cambio significativo, ejecutaremos

```
unix$ mkvers
```

y luego compilaremos el programa. De ese modo el programa podría imprimir (si así se le pide) la fecha que corresponde a la versión del programa.

Este podría ser el script:

```
#!/bin/sh
DATE=`date`

(
    echo -n 'char vers[] = "vers: '
    echo -n "$DATE"
    echo  '";'
) > vers.c
```

La linea que nos interesa es

```
DATE=`date`
```

que es similar a haber escrito

```
DATE="Fri Aug 26 17:12:10 CEST 2016"
```

si es que esa era la fecha.

Lo que hace el shell cuando una línea de comandos contiene "`...`" es ejecutar el comando que hay entre "`...`" y sustituir ese texto de la línea de comandos por la salida de dicho comando. También se dice que el

shell interpola la salida de dicho comando en la línea de comandos. Para conseguir eso, el shell ejecuta código similar a `pipefrom()` y lee la salida del comando. Una vez la ha leído por completo, la utiliza como parte de la línea de comandos y continúa ejecutándola.

Por ejemplo, el comando *seq(1)* se utiliza para contar. Resulta muy útil para numerar cosas. Por ejemplo,

```
unix$ seq 4
1
2
3
4
unix$
```

Como puedes ver en la salida de este comando

```
unix$ echo x 'seq 4' y
x 1 2 3 4 y
unix$
```

el shell ha ejecutado `echo` con "`1 2 3 4`" en la línea de comandos: ha reemplazado el comando entre comillas invertidas por la salida de dicho comando. ¡Y ha reemplazado los "`\n`" en dicha salida por espacios en blanco! Piensa que es una *línea* de comandos y no queremos fines de línea en ella.

Es *muy* habitual utilizar la sustitución de comandos, sobre todo a la hora de programar scripts. Por ejemplo, este libro tenía inicialmente ficheros llamados `ch1.w` para el primer capítulo, `ch2.w` para el segundo, etc.

Podemos crear estos ficheros utilizando un bucle `for` en el shell y la sustitución de comandos. Lo primero que haremos será guardar en una variable los números para los 6 capítulos que pensábamos escribir inicialmente.

```
unix$ caps='seq 6'
unix$ echo $caps
1 2 3 4 5 6
unix$
```

Y ahora podemos ejecutar el siguiente comando que crea cada uno de los ficheros:

```
unix$ for c in $caps
> do
>     echo creating chap $c
>     touch ch$c.w
> done
creating chap 1
creating chap 2
creating chap 3
creating chap 4
creating chap 5
creating chap 6
unix$ echo ch*.w
ch1.w ch2.w ch3.w ch4.w ch5.w ch6.w
unix$
```

Como puedes ver, el comando `for` es de nuevo parte de la sintaxis del shell, similar al comando `if` que vimos anteriormente. En este caso, este comando ejecuta las líneas de comandos entre `do` y `done` (el cuerpo del bucle) para cada uno de los valores que siguen a `in`, haciendo que la variable de entorno cuyo

(c) 2016 Francisco J. Ballesteros

nombre precede a `in` contenga el valor correspondiente en cada iteración. Si no has entendido este párrafo, mira la salida del comando anterior y fíjate en qué es `$c` en cada iteración.

Podríamos haber escrito este otro comando

```
unix$ for c in ‘seq 6‘ ; do
>     echo creating chap $c
>     touch ch$c.w
> done
```

y el efecto habría sido el mismo.

Poco a poco vemos que podemos utilizar el shell para combinar programas existentes para hacer nuestro trabajo. ¡Eso es UNIX!. Recuerda...

- La primera opción es utilizar el manual y encontrar un programa ya hecho que puede hacer lo que deseamos hacer
- La segunda mejor opción es combinar programas existentes utilizando el shell para ello
- La última opción es escribir un programa en C para hacer el trabajo.

7. Pipes con nombre

En ocasiones deseamos poder conectarnos con la entrada/salida de un proceso *después* de que dicho proceso comience su ejecución. Un pipe resulta útil cuando podemos crearlo antes de crear un proceso, y hacer que dicho proceso "herede" los descriptores del pipe. Pero una vez creado, ya no es posible crear un pipe compartido con el proceso.

Existe otra abstracción en UNIX que consiste en un pipe con un nombre en el sistema de ficheros. Se trata de otro tipo de *i-nodo*, llamado **fifo**. Es posible crear un fifo con la llamada *mkfifo(2)*, que tiene el mismo aspecto que *creat(2)*, pero crea un fifo en lugar de un fichero regular. Igualmente, podemos utilizar el comando *mkfifo(1)* para crearlo.

Veamos una sesión de shell que utiliza fifos. Primero vamos a crear uno en `/tmp/namedpipe`:

```
unix$ mkfifo /tmp/namedpipe
unix$ ls -l /tmp/namedpipe
prw-r--r--  1 nemo  wheel  0 Aug 27 09:03 /tmp/namedpipe
unix$
```

Observa que `ls` utiliza una "`p`" para mostrar el tipo de fichero. El fichero es en realidad un pipe. Si utilizamos *stat(2)*, podemos utilizar la constante `S_IFIFO` para comprobar el campo `st_mode` de la estructura `stat` y ver si tenemos un fifo entre manos.

Una vez creado, el fifo se comporta como un pipe. Todo depende de si lo abrimos para leer o para escribir. Por ejemplo, en esta sesión

```
unix$
unix$ cat /tmp/namedpipe &
[1] 16443
unix$ echo hola >/tmp/namedpipe
hola
[1]+  Done                    cat /tmp/namedpipe
unix$
```

vemos como `cat` comienza a ejecutar y queda bloqueado intentando leer del fifo. Una vez ejecutamos `echo` y hacemos que el shell abra el fifo para escribir, `cat` puede leer. En realidad tenemos a `cat` leyendo del extremo de lectura del pipe y a `echo` escribiendo del extremo de escritura del pipe. Una vez `echo`

(c) 2016 Francisco J. Ballesteros

termina y cierra su salida estándar, `cat` recibe una indicación de fin de fichero (lee `0` bytes) y termina.

Si utilizamos el mismo fifo de nuevo, vemos que funciona de modo similar una vez más:

```
unix$ echo hola > /tmp/namedpipe &
[1] 16462
unix$ cat /tmp/namedpipe
hola
[1]+  Done                    echo hola > /tmp/namedpipe
unix$
```

En este caso, `echo` (en realidad el shell al procesar el "`>`") abre el fifo para escribir y se bloquea hasta que algún proceso lo tenga abierto para leer. Una vez `cat` lee el fichero, `echo` puede escribir y termina.

Igual que sucede con un pipe creado con *pipe(2)*, UNIX se ocupa de bloquear a los procesos que leen y escriben en el pipe para que todo funcione como cabe esperar.

El siguiente programa muestra como podríamos utilizar un fifo para leer comandos. Podríamos utilizar algo similar para dotar a una aplicación de una consola a la que nos podemos conectar abriendo un fifo.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

int
main(int argc, char* argv[])
{
    char buf[1024];
    int fd, nr;

    if(mkfifo("/tmp/fifo", 0664) < 0)
        err(1, "mkfifo");
    for(;;){
        fprintf(stderr, "opening...\n");
        fd = open("/tmp/fifo", O_RDONLY);
        if(fd < 0) {
            err(1, "open");
        }
```

(c) 2016 Francisco J. Ballesteros

```
        for(;;){
            nr = read(fd, buf, sizeof buf - 1);
            if(nr < 0){
                err(1, "read");
            }
            if(nr == 0){
                break;
            }
            buf[nr] = 0;
            fprintf(stderr, "got [%s]\n", buf);
            if(strcmp(buf, "bye\n") == 0){
                close(fd);
                unlink("/tmp/fifo");
                exit(0);
            }
        }
        close(fd);
    }
    exit(0);
}
```

El programa abre una y otra vez `/tmp/fifo` (tras crearlo) y lee cuanto puede del mismo. Si consigue leer "`bye\n`", entiende que queremos que el programa termine y así lo hace.

¡Vamos a ejecutarlo!

```
unix$ ./rdfifo &
[1] 16561
unix$ opening...
```

A la vista de los mensajes, `rdfifo` está en la llamada a `open`. Estará bloqueado hasta que otro proceso abra el fifo para escribir en el mismo. Si hacemos tal cosa

```
unix$ echo hola >/tmp/fifo
got [hola
]
opening...
unix$
```

vemos que `read` lee lo que `echo` ha escrito en el fifo, lo que quiere decir que `open` consiguió abrir el fifo para leer del mismo y que `read` pudo obtener varios bytes. Puede verse también que una siguiente llamada a `read` obtuvo `0` bytes y el programa ha vuelto a intentar abrir el fifo. Eso sucede en cuanto `echo` termina y cierra su descriptor.

Podemos ver esto en esta otra sesión de shell, en la que vamos a escribir varias veces utilizando en mismo descriptor de fichero:

```
unix$ (echo hola ; echo caracola) > /tmp/fifo
got [hola
]
got [caracola
]
opening...
unix$
```

Esta vez, varios `read` han podido leer del fifo en nuestro programa. Cuando el último `echo` termina, se cierra el fifo para escribir, lo que hace que nuestro programa reciba una indicación de EOF e intente

(c) 2016 Francisco J. Ballesteros

reabrirlo de nuevo.

Naturalmente, podemos hacer que nuestro programa termine utilizando

```
unix$ echo bye >/tmp/fifo
got [bye
]
[1]+  Done                    ./8.fifo
unix$
```

8. Señales

Otro mecamismo de intercomunicación de procesos es la posibilidad de enviar mensajes a un proceso dado y la posibilidad de actuar cuando recibimos un mensaje en un proceso. Si estás pensando en la red o en TCP/IP, no nos referimos a eso.

UNIX permite enviar mensajes consistentes en números enteros concretos a un proceso dado. A estos mensajes se los denomina **señales** (*signals* en inglés). Cada mensaje tiene un significado concreto y tiene asociada una acción por defecto.

Cuando un proceso entra al kernel (o sale) UNIX comprueba si tiene señales pendientes y el código del kernel hace que cada proceso procese sus propias señales. Así es como funciona, aunque puedes ignorar este detalle. La recepción de una señal

- puede ignorarse,
- puede hacer que ejecute un manejador para la misma, o
- puede matar al proceso.

En breve vamos a ver qué supone esto en realidad.

El comando de shell capaz de enviar una señal a otro proceso es *kill(1)* y la llamada al sistema para hacerlo desde C es *kill(2)*. Estos son algunos de los valores según indica *kill(1)*:

```
unix$ man kill
...
             Some of the more commonly used signals:
     1       HUP (hang up)
     2       INT (interrupt)
     3       QUIT (quit)
     6       ABRT (abort)
     9       KILL (non-catchable, non-ignorable kill)
     14      ALRM (alarm clock)
     15      TERM (software termination signal)
...
```

Por ejemplo, tras ejecutar

```
unix$ sleep 3600 &
[1] 15984
unix$
```

podemos ejecutar `kill` para enviarle la señal `TERM` (simplemente "`15`") al proceso con pid `15984`:

```
unix$ kill 15984
[1]+  Terminated: 15          sleep 3600
unix$
```

(c) 2016 Francisco J. Ballesteros

Como puedes ver, por omisión, la señal `TERM` hace que UNIX mate el proceso que la recibe (salvo que dicho proceso cambie la acción por defecto para esta señal).

Incluso si un comando pide a UNIX que ignore la señal `TERM`, es imposible ignorar la señal `KILL`. Así pues,

```
unix$ kill -9 15984
```

es más expeditivo que el comando anterior, y pide a UNIX que mate el proceso con pid `15984` en cualquier caso (supuesto que tengamos permisos para hacer tal cosa, claro está).

Cuando UNIX inicia un *shutdown* (el administrador decide apagarlo, o pulsas el botón de encendido/apagado) primero envía la señal `TERM` a todos los procesos. Esto permite a dichos procesos enterarse de que el sistema está terminando de operar y podrían salvar ficheros que necesiten salvar o terminar ordenadamente si es preciso. Aquellos procesos a los que no importa esto simplemente habrán dejado que `TERM` los mate.

Pasados unos segundos, UNIX envía la señal `KILL` a todos los los procesos vivos, lo que efectivamente los mata. Puedes pensar que `TERM` significa "*por favor, muérete*" y que `KILL` es una puñalada trapera por la espalda.

Otra señal que seguramente has utilizado es `INT`. Se utiliza para interrumpir la ejecución de un proceso. Es de hecho la señal que se utiliza cuando pulsas *Control* y la tecla `c` antes de soltar *Control*. *Normalmente decimos* "control-c" o escribimos `^C` para representar esto.

Veámoslo. Pero primero hagamos un programa que podamos modificar luego para ver qué hacen las señales.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done");
    exit(0);
}
```

Este programa duerme una hora tras imprimir un mensaje informando de su *pid*. Además, hemos comprobado si `sleep` ha fallado e imprimimos un mensaje tras dormir. Si lo ejecutamos

```
unix$ signal1
pid 16056 sleeping...
```

vemos que no obtenemos el prompt del shell. El proceso está en `sleep`. Si pulsamos ahora *control-c* vemos que la ejecución del programa se interrumpe.

```
unix$ signal1
pid 16056 sleeping...
^C
unix$
```

Obtenemos igual resultado si desde otro shell ejecutamos

(c) 2016 Francisco J. Ballesteros

```
unix$ kill -INT 16056
```

El programa termina igualmente. Si en cambio le envíamos una señal `TERM`, se nos informa de que el proceso ha terminado, aunque el proceso muere igualmente.

```
unix$ signal1
pid 16102 sleeping...
```

Y ejecutando en otro shell

```
unix$ kill -TERM 16102
```

vemos que en shell original el proceso muere:

```
unix$ signal1
pid 16102 sleeping...
Terminated: 15
unix$
```

Cambiemos nuestro programa para pedirle a UNIX que la acción al recibir la señal `INT` no sea morir, sino ignorar la señal.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    signal(SIGINT, SIG_IGN);
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done");
    exit(0);
}
```

Esto lo hacemos utilizando *signal(3)* para pedir que acción por defecto sea ignorar la señal que indicamos (`SIGINT` simplemente es un `2`). La constante `SIG_IGN` indica que queremos ignorar dicha señal.

Ahora volvemos a ejecutar el programa en un shell y pulsar *control-c* varias veces...

```
unix$ signal2
pid 16058 sleeping...
^C^C
```

Esta vez el programa sigue ejecutando como si tal cosa. Podemos utilizar otro shell para ejecutar

```
unix$ kill  16058
```

y matar el proceso (por omisión, `kill` envía la señal `TERM`).

No es muy amable por parte del programa ignorar la señal de interrupción. Seguramente el usuario acabe frustrado sin poder interrumpir el programa y lo mate de todos modos.

Algo más habitual es utilizar *signal* para pedir que cuando se nos envíe la señal `INT` el programa ejecute un manejador para atender dicha señal. Esto es prácticamente lo que sucede con una interrupción software, pero las señales son una abstracción de UNIX para hablar con los procesos, no son interrupciones

(c) 2016 Francisco J. Ballesteros

realmente.

Esta nueva versión del programa pide a UNIX que cuando el proceso reciba la señal `INT` se ejecute la función `handleint`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

static void
handleint(int no)
{
    fprintf(stderr, "hndlr got %d\n", no);
}

int
main(int argc, char* argv[])
{
    signal(SIGINT, handleint);
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done\n");
    exit(0);
}
```

Si lo ejecutamos y pulsamos *control-c* o utilizamos `kill` desde otro shell para enviar la señal `INT`, esto es lo que sucede:

```
unix$ signal3
pid 16150 sleeping...
^C
hndlr got 2
signal3: sleep: Interrupted system call
done
unix$
```

Lo primero que vemos es que *mientras* el programa estaba dentro de `sleep`, el proceso ha recibido la señal. Considerando la llamada a `signal` que hemos hecho, UNIX ajusta la pila (de usuario) del proceso para que cuando continúe ejecutando ejecute en realidad `handleint`. Por eso vemos que lo siguiente que sucede es que `handleint` imprime su mensaje. Una vez `handleint` (el manejador de la señal) termina, su retorno hace que volvamos a llamar a UNIX. Esto lo ha conseguido UNIX ajustando la pila de usuario, nosotros no hemos hecho nada especial en el código como puedes ver. El kernel hace ahora que el programa continúe por donde estaba cuando recibió la señal. En nuestro caso estábamos dentro de `sleep` y, dado que lo hemos interrumpido, `sleep` falla y retorna `-1`. Puedes ver el mensaje de error que imprime `warn` en nuestro código justo detrás del mensaje que ha impreso el manejador. Por último, el programa escribe `done` y termina.

Es preciso tener cuidado con el código que programamos en los manejadores de señal. Dado que el programa podría estar ejecutando cualquier cosa cuando UNIX hace que se interrumpa y ejecute el manejador de la señal, sólo deberíamos utilizar funciones **reentrantes** dentro de los manejadores de señal. Decimos que una función es *reentrante* si es posible llamarla antes de que una llamada en curso termine. Por ejemplo, las funciones que utilizan variables globales (o almacenamiento *estático* en el sentido de `static` en

(c) 2016 Francisco J. Ballesteros

C) no son reentrantes. En pocas palabras, cuanto menos haga un manejador de señal mejor. La página de manual *sigaction(2)* y la de *signal(3)* suelen tener mas detalles respecto a qué funciones puede utilizarse dentro de un manejador y cuales no.

Cada vez que se llama a `signal` para una señal se cambia el efecto de dicha señal en nuestro proceso. Para hacer que `INT` recupere su comportamiento por defecto tras haber instalado un manejador, podríamos ejecutar la llamada

```
signal(SIGINT, SIG_DFL);
```

y UNIX olvidaría que antes teníamos un manejador instalado. La constante `SIG_DFL` indica que queremos que la señal en cuestión tenga la acción por defecto.

Los manejadores de señal que tenemos instalados (incluyendo si consisten en ignorar la señal) son atributos del proceso. Cada proceso tiene los suyos. Un cambio en un proceso a este respecto no afecta a otros procesos.

La página de manual *signal(3)* detalla la lista completa de señales, qué acción por defecto tienen y si se pueden ignorar o no.

Las llamadas al sistema interrumpidas por una señal suelen devolver una indicación de error y dejan `errno` al valor `EINTR` (normalmente), que significa "*interrupted*".

Pero `read` y `write` tienen un comportamiento especial. Cuando se interrumpen por el envío de una señal, UNIX re-inicia las llamadas al sistema tras la interrución (y tras la ejecución del manejador si lo hay). Naturalmente, salvo que la acción de la señal en cuestión sea matar al proceso. Eso quiere decir que nuestro programa continuaría ejecutando `read` si es que `read` se ha interrumpido, tras ejecutar el manejador que hemos instalado antes para `INT`.

Para cambiar este comportamiento, podemos utilizar *siginterrupt(3)* y pedir que una señal interrumpa las llamadas al sistema que pueden re-iniciarse tras una señal, esto es, `read` y `write` principalmente. Por ejemplo, ejecutando

```
siginterrupt(SIGINT, 1);
```

incluso `read` se interrumpirá y dejará en `errno` el valor `EINTR`. En cambio, tras

```
siginterrupt(SIGINT, 0);
```

las llamadas a `read` interrumpidas volverán a reiniciarse automáticamente.

Dos señales útiles son `USR1` y `USR2`. Suelen estar disponibles para lo que nos plazca. Un uso habitual es hacer que el programa vuelque información de depuración, por ejemplo, cuando se le envíe `USR1`, o que relea la configuración. También suele utilizarse la señal `HUP` para que el programa relea la configuración, aunque esta señal no es precisamente para eso.

9. Alarmas

Podemos combinar las señales con un temporizador que tiene cada proceso para implementar time-outs. El temporizador es otra abstracción que suministra UNIX para cada proceso. Podemos programarlo para que envíe la señal `ALRM` pasado un tiempo, e instalar el manejador que queramos para dicha señal.

Veamos un programa que hace esto:

(c) 2016 Francisco J. Ballesteros

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>
#include <signal.h>

static void
tout(int no)
{
    fprintf(stderr, "interrupted\n");
}

int
main(int argc, char* argv[])
{
    char buf[1024];
    int nr;

    signal(SIGALRM, tout);
    fprintf(stderr, "> ");
    siginterrupt(SIGALRM, 1);    // try with 0
    alarm(15);
    nr = read(0, buf, sizeof buf - 1);
    alarm(0);
    if(nr > 0){
        write(1, buf, nr);
    } else {
        warn("read");
    }
    exit(0);
}
```

Si lo ejecutamos y escribimos una línea antes de que pasen 15 segundos esto es lo que sucede:

```
unix$ 8.alrm
> hola
hola
unix$
```

El programa termina la llamada a `read` y escribe lo que ha leído. Pero si lo ejecutamos y pasan 15 segundos...

```
unix$ 8.alrm
> interrupted
8.alrm: read: Interrupted system call
unix$
```

Primero se interrumpe `read`, y ejecuta el manejador `tout` que hemos instalado (que imprime "`interrupted`"). Luego `read` termina indicando como error `EINTR`, y el programa continúa.

La primera llamada a `alarm` instala el temporizador para que envíe la señal `ALRM` pasados 15 segundos. La segunda llamada a `alarm` (tras `read`) cancela el temporizador.

Observa la llamada a `siginterrupt`, ¿Qué sucede si la quitamos?

Una última advertencia. Los temporizadores han de usarse con sumo cuidado. De hecho, si es posible, es mejor no utilizarlos. Hacen los programas impredecibles y difíciles de depurar. Por ejemplo, ¿Qué pasa si el

(c) 2016 Francisco J. Ballesteros

usuario tardó 16 segundos en escribir su línea? ¿Por qué falla el programa en esos casos? Tal vez sería mejor dejar el programa bloqueado leyendo en nuestro caso...

10. Terminales y sesiones

El *terminal* es el dispositivo en UNIX que representa la consola, ventana o acceso remoto al sistema utilizado para acceder al sistema. Ya hemos visto que `/dev/tty` representa el terminal en cada proceso y que hay otros dispositivos habitualmente llamados `/dev/tty*` o `/dev/pty*` que representan terminales concretos.

Nos interesa ser conscientes de este dispositivo puesto que es capaz de enviar señales a los procesos y además suministra la entrada/salida en la mayoría de programas interactivos. La idea es que el terminal controla los procesos que lo utilizan y, por ejemplo, es capaz de enviar la señal `INT` si se pulsa *control-c* y es capaz de enviar la señal `HUP` si el usuario sale del terminal (cierra la ventana, desconecta la conexión de red que ha utilizado para conectarse, lo hace un log-out de la consola).

La abstracción es sencilla: UNIX agrupa los procesos en **grupos de procesos** y cada uno pertenece a una **sesión**. A su vez, cada sesión tiene un **terminal de control** (en realidad cada proceso lo tiene).

Por ejemplo, si abrimos una ventana con un shell obtenemos un nuevo terminal. Podemos ver qué terminal estamos utilizando con el comando *tty(1)*. Por ejemplo, en una ventana tenemos

```
unix$ tty
ttys002
unix$
```

y en otra

```
unix$ tty
ttys007
unix$
```

tenemos un terminal distinto.

Pues bien, la *sesión* es la abstracción que representa la sesión en el sistema en cada uno de estos terminales. En nuestro ejemplo (y si no hay ninguna otra consola ni conexión remota, lo cual no es cierto) tendríamos dos sesiones.

Dentro de una sesión tenemos ejecutando un shell y podemos ejecutar una línea de comandos tal como

```
unix$ ls | wc -l
```

o cualquier otra. En este caso, tanto `ls` como `wc` terminarán perteneciendo al mismo *grupo* de procesos. Y si ejecutamos

```
unix$ (sleep 3600 ; echo hi there) &
unix$ ls | wc -l
```

tendremos dos grupos de procesos. La intuición es que podemos utilizar los grupos para agrupar los procesos (de una línea de comandos, por ejemplo).

Así pues tenemos procesos agrupados en grupos que están agrupados a su vez en sesiones. Cada una de estas abstracciones es simplemente otro tipo de datos más implementado por el kernel y como todo lo demás tendrá su propio record con los atributos que deba tener dentro del kernel. Una vez más es tan sencillo como eso.

Un proceso puede iniciar una nueva sesión llamando a

(c) 2016 Francisco J. Ballesteros

```
setsid();
```

lo que a su vez hace que también cree un nuevo grupo de procesos y se convierta en su líder. La función *setpgid(2)* es la que se utiliza para hacer que un proceso cree un nuevo grupo de procesos y pase a ser su líder (aunque seguirá dentro de la misma sesión).

Es poco habitual que tengas que llamar a estas funciones. Podrías desear que tu programa se deshaga del terminal de control para que continue ejecutando como un proceso en background sin que ningún terminal le envíe señal alguna (para que ejecute como un **demonio**, que es como se conoce a estos procesos). Esto sucederá si estás programando un servidor o cualquier otro programa que se desea que continue su ejecución independientemente de la sesión.

Pero incluso en este caso, lo normal es llamar a *daemon(3)* que crea un proceso utilizando *fork(2)* y hace que su directorio actual sea `"/"` y que no tenga terminal de control. De ese modo puede ejecutar sin molestar. Eso si, si dicho programa vuelve a utilizar un terminal para leer o escribir, seguramente adquiera dicho terminal como terminal de control. Aunque los detalles exactos dependen del tipo de UNIX concreto que utilices. Consulta tu manual.

El comportamiento del terminal está descrito en *tty(4)* y puede cambiarse. Desde el shell disponemos del comando *stty(1)* (*set tty*) y desde C disponemos de una llamada al sistema *ioctl(2)* que es en realidad una forma de efectuar múltiples llamadas que no tienen llamada al sistema propia (simplemente se utiliza una constante para indicar qué operación de control de Entrada/Salida queremos hacer y otros argumentos empaquetan los parámetros/resultados de la llamada en un record).

Por ejemplo, estos son los ajustes de nuestro terminal

```
unix$ stty -a
speed 9600 baud; rows 24; columns 58; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
eol = M-^?; eol2 = M-^?; swtch = <undef>; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr
icrnl ixon -ixoff -iuclc -ixany imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0
cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase
-tostop -echoprt echoctl echoke
unix$
```

Conviene no intimidarse por la cantidad de ajustes. Hemos mostrado todos utilizando el flag `-a` de `stty`. Los terminales eran artefactos mas o menos simples cuando UNIX se hizo allá por los 70, pero hoy día acarrean toda una historia de dispositivos pintorescos.

Una cosa que vemos es que la opción `intr` tiene como valor `^C`. Esto quiere decir que si pulsamos *control-c* el terminal envía la señal *INT* al grupo de procesos que está ejecutando en **primer plano** o **foreground**. Dicho de otro modo, a los involucrados en la línea de comandos que estamos ejecutando en el momento de pulsar `^C`.

Aquellos procesos que hemos ejecutado el **background** o en **segundo plano**, esto es, aquellos que hemos ejecutado con un `"&"` en su línea de comandos pertenecen a otro grupo de procesos y no reciben la señal `INT` cuando pulsamos `^C`.

Podemos cambiar la combinación de teclas que produce la señal de interrumpir el grupo en primer plano.

(c) 2016 Francisco J. Ballesteros

Esto podemos hacerlo por ejemplo como en

```
unix$ stty intr ^h
unix$
```

Aquí hemos escrito literalmente "^h", no hemos pulsado *control-h*. Tras la ejecución de este comando, *control-c* no interrumpe la ejecución de ningún comando en este terminal. En cambio, *control-h* sí que puede utilizarse para interrumpir al grupo que está ejecutando en foreground.

Para restaurar los valores por defecto o dejarlos en un estado razonable podemos ejecutar

```
unix$ stty sane
```

y volvemos a utilizar ^C para interrumpir.

Otro ajuste interesante es el flag de echo. Si ejecutamos

```
unix$ ssty -echo
```

(quitar el eco) veremos que en las siguientes líneas de comandos no vemos nada de lo que escribimos. No obstante, si escribimos el nombre de un comando y pulsamos *enter* vemos que UNIX lo ejecuta normalmente.

```
unix$ stty -echo
unix$ Sat Aug 27 11:23:24 CEST 2016
```

El programa que escribe en pantalla lo que nosotros escribimos en el teclado es el terminal, y claro, si le pedimos que no haga eco de lo que escribimos, deja de hacerlo. Podría ser útil para leer contraseñas y para alguna otra cosa...

Para dejar el terminal en su estado normal podemos ejecutar

```
unix$ stty echo
```

aunque no veamos por el momento lo que escribimos. Aunque si es una ventana, es más simple cerrarla y abrir otra.

10.1. Modo crudo y cocinado

Un ajuste importante es el modo de **disciplina de línea** del terminal. Se trata de un parámetro que puede tener como valor *modo crudo* (o *raw*) o bien *modo cocinado* (*cooked*).

Como ya sabes el terminal procesa caracteres según los escribes pero no suministra esos caracteres a ningún programa que lea del terminal hasta que pulsas *enter*. A esto se le llama *modo cocinado*. El terminal "cocina" la línea para permitir que puedas editarla.

Pero en algunas ocasiones querrás procesar directamente los caracteres que escribe el usuario (por ejemplo, si estás implementando un juego o un editor y quieres atender cada carácter por separado). El *modo crudo* sirve justo para esto. Si lo activas, el terminal no cocina nada y se limita a darte los caracteres según estén disponibles. Naturalmente, no podrás borrar y, de hecho, el carácter que utilizabas para borrar pasará a comportarse como cualquier otro carácter.

Con que sepas que estos modos existen tenemos suficiente. Puedes utilizar el manual y las llamadas que hemos mencionado recientemente para aprender a utilizar estos modos, aunque es muy poco probable que lo necesites.

Capítulo 6: Usando el shell

(c) 2016 Francisco J. Ballesteros

1. Comandos como herramientas

En UNIX la idea es combinar programas ya existentes para conseguir hacer el trabajo. Actualmente, por desgracia, hay muchos usuarios de UNIX que lo han olvidado y corren a implementar programas en C u otros lenguajes para tareas que ya se pueden resolver con los programas existentes.

Por ejemplo, quizá conozcas un programa llamado *head(1)* que imprime las primeras líneas de un fichero. Si recuerdas que *seq(1)* escribe líneas con enteros hasta el valor indicado, verás que en

```
unix$ seq 10 | head -2
1
2
unix$
```

`head` ha impreso las primeras dos líneas aunque `seq` escriba 10 en este caso. Pues resulta que no era preciso programar `head`, dado que con un editor de *streams* (texto según fluye por un pipe) ya era posible imprimir las primeras *n* líneas. Baste un ejemplo:

```
unix$ seq 10 | sed 2q
1
2
unix$
```

Se le pide educadamente a *sed(1)* que termine tras imprimir las primeras dos líneas sin hacer ninguna edición en ellas y ya lo tenemos.

Saber utilizar y combinar los comandos disponibles en UNIX te resultará realmente útil y ahorrará gran cantidad de tiempo. Además, dado que estos programas ya están probados y depurados evitarás bugs innecesarios. Simplemente utiliza el manual (¡Y aprende a buscar en él!) para localizar comandos que hagan o todo o parte del trabajo que deseas hacer. Combina varios de ellos cuando no sea posible efectuar el trabajo on uno sólo de ellos, y recurre a programar una nueva herramienta sólo como último recurso. Los comandos de unix son una caja herramientas, ¡úsala!.

La utilidad del shell es, además de permitirnos escribir comandos simples, permitirnos combinar programas ya existentes para hacer otros nuevos. Ya lo hemos estado haciendo anteriormente, pero ahora vamos a dedicarle algo de tiempo a ver cómo programar utilizando el shell. Recuerda que *sh(1)* está disponible en todos los sistemas UNIX, por lo que aprender a utilizarlo te permitirá poder combinar comandos y programar scripts en todos ellos.

2. Convenios

Para facilitarte el trabajo a la hora de utilizar el shell, deberías ser sistemático y seguir tus propios convenios. Por ejemplo, si las funciones en C las escribimos siempre como en

```
int
myfunc(int arg)
{
    ...
}
```

entonces sabemos que todas las líneas con un identificador a principio de linea seguido de un "`(`" está definiendo una función.

El comando *egrep(1)* imprime líneas que encajan con una expresión (como veremos más adelante). Si hemos sido sistemáticos con nuestro convenio para programar funciones, podríamos pedirle que escriba todas las líneas que declaran funciones en un directorio dado. No te preocupes por cómo se usa `egrep` por el momento, presta atención a cómo un simple convenio nos permite trabajar de forma efectiva.

(c) 2016 Francisco J. Ballesteros

Podemos utilizar el comando *egrep(1)* para ver qué funciones definimos y en qué ficheros y líneas:

```
unix$ egrep -n '^[a-z0-9A-Z_]+\(' *.c
alrm.c:11:tout(int no)
alrm.c:17:main(int argc, char* argv[])
broke.c:8:main(int argc, char* argv[])
fifo.c:14:main(int argc, char* argv[])
...
```

O para contar cuántas funciones definimos...

```
unix$ egrep -n '^[a-z0-9A-Z_]+\(' *.c | wc -l
      24
unix$
```

Programando en un entorno integrado de desarrollo (IDE o *integrated development environment*) puedes hacer este tipo de cosas pulsando botones con el ratón. ¡Pero ay de ti si quieres hacer algo que no esté disponible en tu IDE! Con frecuencia notarás que estás haciendo trabajo repetitivo y mecánico, ¡incluso si utilizas un IDE en lugar de un editor sencillo! Cuando te suceda eso, recuerda que seguramente puedas programar un script que haga el trabajo por ti.

El shell es muy bueno manipulando ficheros, bytes que proceden de la salida de un comando y texto en general. Igual sucede con los comandos que manipulan ficheros de texto en UNIX. Para programar utilizando el shell hay que pensar que no estamos utilizando C e intentar escribir comandos que operen sobre todo un fichero a la vez o sobre todo un flujo de bytes a la vez. La idea es ir adaptando los datos que tenemos a los que queremos tener. Otra estrategia es adaptar los datos que tenemos para convertirlos en comandos que, una vez ejecutados, produzcan el efecto que deseamos. Iremos viendo esto poco a poco con los ejemplos que siguen.

En muchos casos es posible que estés intentando resolver problemas que has creado tu mismo. Lo mejor en la mayoría de los casos suele ser intentar no crear los problemas en lugar de resolverlos. Baste un ejemplo.

Supón que estás programando una aplicación que requiere de un fichero de configuración. Tradicionalmente estos ficheros suelen guardarse en el directorio `$HOME` con nombres que comienzan por `"."`, pero eso nos da igual en este punto. Ya sea por seguir la moda o por no saber cómo leer líneas y cómo partirlas en palabras, quizá programemos la aplicación utilizando XML como formato para el fichero de configuración. ¡Qué gran error! Al menos, en los casos en que nuestro fichero de configuración necesite información tan simple como

```
shell /bin/sh
libdir /usr/lib
```

para indicar qué shell queremos que ejecute y qué directorio queremos que utilice para almacenar sus librerías (por ejemplo). El lugar de un fichero tan sencillo como el que hemos mostrado, quizá termines con algo como

(c) 2016 Francisco J. Ballesteros

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE MyConfig SYSTEM "Crap_Config.dtd">
<Crap_Config>
    <Parameters>
        <Param>
            <Name>shell</Name>
            <Value>/bin/sh</Value>
        </Param
        <Param>
            <Name>libdir</Name>
            <Value>/usr/lib</Value>
        </Param
    </Parameters>
</Crap_Config>
```

Es cierto que tienes librerías ya programadas para la mayoría de lenguajes que saben entender ficheros en este formato, leerlos y escribirlos. Pero eso no es una excusa si el fichero en cuestión guarda información que puedas tabular (líneas de texto con campos en cada línea). En realidad, ni siquiera es una excusa si necesitas guardar una estructura de tipo árbol que sea sencilla, como por ejemplo este fichero:

```
/
    home
        nemo
    bin
    tmp
```

que podría indicar con qué partes de un árbol de ficheros queremos trabajar.

Pero volvamos a nuestro fichero de configuración. Supón que deseas comprobar que para todos los usuarios que utilizan tu aplicación en el sistema, los shells que han indicado existen y son ejecutables y los directorios existen. ¿Cómo lo harás? Con el formato sencillo para el fichero bastaría hacer algo como

```
unix$ shells=`egrep '^shell' /home/*/.myapp | cut -f2`
```

para tener en `$shells` la lista de shells y luego podemos utilizar esta variable y un bucle `for` del shell para comprobar que existen. Si has utilizado XML también podrías programar un script para hacerlo, pero te resultará mucho más difícil. Te has buscado un problema que no tenías.

Pero vamos a ver cómo usar el shell para hacer este tipo de cosas...

3. Usando expresiones regulares

Vamos a resolver el problema que teníamos. Supongamos que tenemos nuestra aplicación `myapp` con su fichero de configuración en `$HOME/.myapp` para cada usuario, y que su aspecto es como habíamos visto:

```
shell /bin/sh
libdir /usr/lib
bindir /usr/bin
logdir /log
```

Vamos a hacer un script llamado `chkappconf` que compruebe la configuración para todos los usuarios del sistema. Por el momento vamos a usar un fichero llamado `config` que tiene justo el contenido que mostramos arriba como ejemplo de fichero de configuración. Luego es fácil trabajar con los ficheros de verdad. La idea es procesar los ficheros *enteros*, poco a poco, e ir generando nuevos "ficheros" como salida de comandos que procesan los datos que tenemos. En realidad, generamos bytes que fluyen por un pipe.

En este caso, vamos a necesitar utilizar *grep(1)*, o, concretamente *egrep(1)*. Este comando lee su entrada, o

(c) 2016 Francisco J. Ballesteros

ficheros indicados en la línea de comandos, línea a línea y escribe aquellas que encajan con la expresión que le damos. Por ejemplo,

```
unix$ egrep dir config
libdir /usr/lib
bindir /usr/bin
logdir /log
```

El primer parámetro es la expresión que buscamos y el segundo es el fichero en que estamos buscando.

Otro ejemplo:

```
unix$ seq 15 | egrep 1
1
10
11
12
13
14
15
unix$
```

En este caso `egrep` lee de la entrada estándar, dado que no hemos indicado fichero alguno.

Las expresiones de `egrep` son muy potentes. Son de hecho un lenguaje denominado **expresiones regulares** que se utiliza en diversos comandos de UNIX, por lo que resulta muy útil aprenderlo. Mirando la página *egrep(1)* vemos hacia el final...

```
unix$ man egrep
...
SEE ALSO
     ed(1), ex(1), gzip(1), sed(1), re_format(7)
...
```

Y precisamente *re_format(7)* en este sistema documenta las expresiones regulares. Esto lo hemos hecho en un sistema OS X. Si usamos un sistema Linux podemos hacer la misma jugada

```
unix$ man egrep
...
SEE ALSO
awk(1),  cmp(1), diff(1), find(1), sed(1), sort(1),
       glob(7), regex(7).
...
```

y vemos que *regex(7)* documenta sus expresiones regulares. Las que usamos en este curso funcionan en ambos.

Una expresión regular es un string que describe otros strings. Decimos que un string encaja con la expresión si *contiene* uno de los strings que describe la expresión. Podemos definir las expresiones recursivamente como sigue:

- Cualquier carácter normal que no forma parte de la sintaxis de expresiones regulares encaja con el mismo. Por ejemplo, `a` describe el string `a`.
- La expresión "`.`" describe con cualquier carácter, pero sólo uno. Por ejemplo, `.` puede ser tanto `a` como `b`, pero no el string vacío ni tampoco on `ab`.
- Una expresión regular *r0* seguida de otra *r1* describe los strings que tienen un prefijo descrito por *r0* seguido de otro descrito por *r1*. Por ejemplo `ab` describe el string `ab` puesto que `a` puede ser `a` y `b` puede ser `b`. Igualmente, `.b` describe `ab` `bb` pero no `ba`.

(c) 2016 Francisco J. Ballesteros

- Si *r0* y *r1* son dos expresiones regulares, La expresión *r0*|*r1* describe los strings que describe alguna de las expresiones *r0* y *r1* (o que describen las dos).
- Si tenemos una expresión *r*, entonces *r*`*` describe los strings "" (la cadena vacía), los que describe *r*, los de *rr*, los de *rrr*, etc. (Repetimos la expresión cualquier número de veces, posiblemente ninguna).
- Si tenemos una expresión *r*, entonces *r*`+` es lo mismo que *r*`(`*r*`)*`. Esto es, *r* una o más veces.
- Si tenemos una expresión *r*, entonces *r*`?` es lo mismo que `(`*r*`)|()`. Esto es, *r* una o ninguna vez.
- `^` representa el principio del string en que buscamos encajes de la expresión regular. Por ejemplo, `^a` encaja con `ab` pero no con `ba`.
- `$` representa el final del string en que buscamos encajes de la expresión regular. Por ejemplo, `a$` encaja con `ba` pero no con `ab`.
- `\`*c* quita el significado especial a *c*, de tal modo que podemos utilizar caracteres que forman parte de la sintaxis de expresiones regulares como caracteres normales. Por ejemplo, `\\` encaja con `\` y `\.c` encaja con `.c` pero no con `ac`. En cambio `.c` encaja también con `ac`.
- `(`*r*`)` permite agrupar una expresión y describe los strings descritos por *r*. Por ejemplo, `(a|b)(x|y)` describe `ax` pero no `ab`.
- `[`...`]` describe cualquiera de los caracteres entre los corchetes. Y es posible escribir rangos como en `[a-c]` (de la `a` a la `c`). Por ejemplo, `[a-zA-Z0-9_]` es cualquier letra minúscula o mayúscula o dígito o bien "`_`". ¡Pero cuidado aquí con caracteres como "`ñ`"!
- `[^`...`]` describe cualquiera de los caracteres no descritos por `[`...`]`. Por ejemplo, `[0-9]` es cualquier carácter que no sea un dígito.

Veamos algunos ejemplos utilizando `seq` para usar `egrep` en su salida. Primero, buscamos un `1` seguido de un carácter:

```
unix$ seq 15 | egrep 1.
10
11
12
13
14
15
unix$
```

Ahora un carácter seguido de un `1`:

```
unix$ seq 15 | egrep .1
11
unix$
```

O bien `11` o bien `12`:

```
unix$ seq 15 | egrep '11|12'
11
12
unix$
```

Y tambien podemos combinar expresiones más complejas del mismo modo:

(c) 2016 Francisco J. Ballesteros

```
unix$ seq 15 | egrep '.2|1.'
10
11
12
13
14
15
unix$
```

Un `1` y cualquier cosa:

```
unix$ seq 15 | egrep '1.*'
1
10
11
12
13
14
15
unix$
```

¡Ojo al `.*`! Si usáramos `1*` entonces veríamos todas las líneas dado que todas contienen el string vacío y que `1*` encaja con el string vacío. Pero podríamos pedir las líneas que son cualquier número de unos:

```
unix$ seq 15 | egrep '^1*$'
1
11
unix$
```

Un `1` o el principio del texto y luego un `2` o un `3`:

```
unix$ seq 15 | egrep '(1|^)(2|3)'
2
3
12
13
unix$
```

Aunque tal vez sería mejor

```
unix$ seq 15 | egrep '(1|^)[23]'
2
3
12
13
unix$
```

para conseguir el mismo efecto.

Ahora cualquier línea que use sólo cualquier carácter menos los del `2` al `8`:

```
unix$ seq 15 | egrep '^[^2-8]*$'
1
9
10
11
```

Líneas que tengan `3` una o más veces:

(c) 2016 Francisco J. Ballesteros

```
unix$ seq 15 | egrep '3+'
3
13
unix$
```

Quizá un 1 y un 3:

```
unix$ seq 15 | egrep '1?3'
3
13
unix$
```

Bueno, ya conocemos las expresiones regulares y podemos utilizar `egrep` para obtener las líneas de nuestro fichero de configuración que se refieren al shell:

```
unix$ egrep '^shell ' config
shell /bin/sh
unix$
```

Y también las que se refieren a directorios:

```
unix$
unix$ egrep '^[a-z]+dir ' config
libdir /usr/lib
bindir /usr/bin
logdir /log
unix$
```

Nótese que utilizar aquí "`dir`" como expresión habría sido seguramente un error. Líneas que contengan algo como "/opt/bindir/ksh" habrían salido y no tienen por qué ser las que buscamos en este caso.

4. Líneas y campos

Tenemos toda una plétora de comandos que saben trabajar con ficheros suponiendo que tienen líneas y que cada una tiene una serie de campos. Muchos de los comandos que trabajan con líneas no requieren siquiera que tengan campos o cualquier otra estructura predeterminada. Vamos a continuar nuestro script de ejemplo centrándonos ahora en este tipo de tarea: procesar líneas y campos.

Teníamos ya, para nuestro fichero

```
shell /bin/sh
libdir /usr/lib
bindir /usr/bin
logdir /log
```

un comando capaz de escribir las líneas que corresponden a directorios:

```
unix$
unix$ egrep '^[a-z]+dir ' config
libdir /usr/lib
bindir /usr/bin
logdir /log
unix$
```

Ahora nos gustaría tener la lista de directorios mencionados en dichas líneas. Lo primero que tenemos que hacer es pensar que la entrada que tenemos es un fichero con líneas que tienen dos campos separados por

(c) 2016 Francisco J. Ballesteros

blanco. En tal caso basta con utilizar un programa capaz de escribir el segundo campo. Hay muchas formas de hacer esto. Una es utilizar *cut(1)*. El flag `-f` de `cut` admite como argumento el número de campo que se desea, comenzando a contar desde `1`.¡Pero atención!

```
unix$ egrep '^[a-z]+dir ' config | cut -f2
libdir /usr/lib
bindir /usr/bin
logdir /log
unix$
```

Por omisión `cut` utiliza el tabulador como carácter separador de campos y, en nuestro caso, teníamos un espacio en blanco y no un tabulador separando los campos. En realidad queremos

```
unix$ egrep '^[a-z]+dir ' config | cut '-d ' -f2
/usr/lib
/usr/bin
/log
unix$
```

que utiliza la opción `-d` de `cut` para indicarle que el carácter escrito tras la opción es el que deseamos utilizar como separador. ¡Ya tenemos una lista de directorios!

No obstante, `cut` no es la mejor opción en la mayoría de los casos. Pensemos en obtener la lista de dueños de los ficheros del directorio actual... Sabiendo que podemos usar un listado largo en `ls` como en

```
unix$ ls -l config
-rw-r--r--  1 nemo  staff  58 Aug 28 11:27 config
```

podríamos intentar usar algo como

```
unix$ ls -l config | cut -f3
-rw-r--r--  1 nemo  staff  58 Aug 28 11:27 config
```

¡Pero `cut` escribiría todos los campos! El problema de nuevo es el separador entre un campo y otro.

El programa que menos problemas suele dar para seleccionar campos es *awk(1)*. Es un lenguaje de programación para manipular ficheros con aspecto de tabla, pero tiene muchos programas de una sóla línea que resultan útiles. En este caso,

```
unix$ ls -l config | awk '{print $3}'
nemo
```

consigue el efecto que buscamos. El programa significa "*en todas las líneas, imprime el tercer campo*".

Lo bueno de `awk` es que utiliza una expresión regular como separador de campos. Y puedes cambiarla escribiendo la que quieras como argumento de la opción `-F`:

```
awk '-F[: ]+' '{print $3}'
```

imprime el tercer campo suponiendo que un campo está separado de otro por uno o mas caracteres que sean bien un espacio en blanco o bien dos puntos.

Volviendo a nuestro problema, una vez tenemos los directorios mencionados en nuestro fichero de configuración, estaría bien tenerlos ordenados y sin tener duplicados en la lista. Para conseguirlo podemos utilizar el comando *sort(1)* que sabe ordenar líneas utilizando campos como clave (o la línea entera). Otro comando relacionado es `uniq(1)`, que elimina duplicados de una entrada ya ordenada. Así pues,

(c) 2016 Francisco J. Ballesteros

```
unix$ egrep '^[a-z]+dir ' config | awk '{print $1}' | sort | uniq
bindir
libdir
logdir
unix$
```

escribe los directorios ordenados y sin duplicados.

Esto resulta útil también al recolectar nombres o valores que aparecen en cualquier otro sitio. Por ejemplo, para obtener la lista de dueños de ficheros en el directorio actual:

```
unix$ ls -l * | awk '{print $3}' | sort -u
nemo
unix$
```

El flag `-u` de `sort` hace lo mismo que `uniq`. ¡Ya tenemos algo que podríamos incluso guardar en una variable de entorno!

```
unix$ owners=`ls -l * | awk '{print $3}' | sort -u`
unix$ echo $owners
nemo
unix$
```

De hecho, vamos a hacer justo con nuestros directorios, y ya podemos ver si existen o no...

```
unix$ dirs=`egrep '^[a-z]+dir ' config | awk '{print $1}' | sort | uniq`
unix$ for d in $dirs ; do
>    if test ! -d $d ; then
>        echo no dir $d
>    fi
>    done
unix$
```

¡Bien, todos los directorios existen!, ¡Problema resuelto!

Recuerda que es el shell el que escribe los ">" para indicarnos que es preciso escribir más líneas para completar el comando. Hemos usado el bucle `for` que vimos anteriormente y que ejecuta los comandos entre `do` y `done` para cada palabra que sigue a `in`. En cada iteración, la variable cuyo nombre precede a `in` toma como valor cada una de las palabras que siguen a `in`.

También hemos utilizado el comando `if` que ejecuta en este caso

```
test ! -d $d
```

como condición y, de ser cierta, ejecuta los comandos en el `then`. Aunque no lo hemos necesitado, es posible escribir

```
if ....
then
    ...
else
    ...
fi
```

como comando.

El comando `sort` merece que le dediquemos algo más de tiempo. Sabe ordenar la entrada asumiendo que son cadenas de texto o bien ordenarla segun su valor numérico (y de algunas otras formas). Por ejemplo,

(c) 2016 Francisco J. Ballesteros

```
unix$ seq 10 | sort
1
10
2
3
4
5
6
7
8
9
```

no produce el efecto que podrías esperar. Ahora bien...

```
unix$ seq 10 | sort -n
1
2
3
4
5
6
7
8
9
10
```

con la opción `-n`, `sort` ordena numéricamente.

Sea numérica o alfabéticamente, podemos pedir una ordenación en orden inverso:

```
unix$ seq 5 | sort -nr
5
4
3
2
1
unix$
```

Aquí utilizamos ambos flags para indicar que se desea una ordenación numérica y además en orden inverso.

Además podemos pedir a `sort` que utilice como clave para ordenar sólo alguno de los campos, por ejemplo

```
sort -k3,5
```

ordena utilizando los campos del 3 al 5 (contando desde uno). Consulta su página de manual en lugar de recordar todo esto.

¿Recuerdas el comando *du(1)*? !Podemos ver dónde estamos gastando el espacio en disco!

```
unix$ du -s * | sort -nr | sed 5q
23824    zx-spe
6088    wr_pic1.eps
6088    inkdump.eps
6088    clive_pic3.eps
5768    zx
```

Primero, hacemos que `du` liste el total (opción `-s`, *summary*) para todos los ficheros y directorios. Ordenamos entonces su salida numéricamente en orden inverso y nos quedamos con las 5 primeras líneas. Son los

5 ficheros o directorios en que estamos usando más disco. Ahora podemos pensar qué hacemos con ellos si necesitamos espacio...

¿Y si queremos los que menos espacio usan? Podríamos quedarnos con las últimas 5 líneas, por ejemplo:

```
unix$ du -s * | sort -nr | tail -5
```

utiliza *tail(1)* que escribe las últimas *n* líneas de un fichero. Aunque habría sido mejor no invertir la ordenación en sort:

```
unix$ du -s * | sort -n | sed 5q
```

El comando `tail` tiene otro uso para imprimir las últimas líneas pero empezando a contar desde el principio. Esto es útil, por ejemplo, para eliminar las primeras líneas de un fichero:

```
unix$ seq 5 | tail +3
3
4
5
unix$
```

También es posible seleccionar determinados campos y/o reordenarlos. Por ejemplo, con la opción `-l`, `ps` produce un listado largo...

```
unix$ man ps
...
     -l      Display information associated with the following keywords:
             uid, pid, ppid, flags, cpu, pri, nice, vsz=SZ, rss, wchan,
             state=S, paddr=ADDR, tty, time, and command=CMD.
...
```

Para cada item, `ps` produce una columna separada de las demás por espacio en blanco. Así pues podemos utilizar `awk` para imprimir el pid, tamaño de la memoria física (*resident set size*, o rss) y nombre de los procesos utilizando

```
unix$ ps -l | awk '{printf("%s\t%s\t%s\n", $2, $15, $9);}'
PID    CMD    RSS
448    -bash    372
519    acme    28
526    (fontsrv)    0
527    acme    104
528    acme    5576
534    9pserve    24
...
```

Como puedes ver, `awk` dispone de `printf`. Dicha función se utiliza prácticamente como la de C. En nuestro caso optamos por imprimir las columnas 2, 15, y 9 en ese orden. Utilizar `cut` sería más complicado dado que hay múltiples blancos entre un campo y el siguiente.

¿Qué proceso está utilizando más memoria? Bastar ordenar numéricamente por el tercer campo, de mayor a menor y quedarse con la primera línea.

```
unix$ ps -l | awk '{printf("%s\t%s\t%s\n", $2, $15, $9);}' | \
>    sort -nr -k3 | sed 1q
91136    acme    52920
unix$
```

(c) 2016 Francisco J. Ballesteros

Parece que `acme`, con el pid 91136.

5. Funciones y otras estructuras de control

Aún tenemos pendiente escribir nuestro script para procesar los ficheros de configuración de nuestra aplicación y ver si todos son correctos.

Lo primero que deberíamos hacer es pensar en qué opciones y argumentos queremos admitir en nuestro script. Por ejemplo, un posible uso podría ser

```
chkappconf [-sd] [fich...]
```

Hemos utilizado la sintáxis que vemos en la synopsis de las páginas de manual:

- Los flags `-d` y `-s` pueden utilizarse opcionalmente y permiten comprobar sólo los directorios utilizados en los ficheros de configuración o sólo los shells mencionados. Si no indicamos ninguno de estos flags queremos que se comprueben ambas cosas.
- Si se indican nombres de fichero (uno o más) como argumento entonces haremos que se comprueben sólo dichos ficheros, en caso contrario comprobaremos todos los ficheros en los directorios `$HOME` de todos los usuarios.

Sabemos que `$*` corresponde al vector de argumentos en un script, excluyendo el nombre del script (`argv[0]`, que sería `$0`). Una posibilidad es procesar dicha lista de argumentos mientras veamos que comienzan por "`-`", y luego veremos cómo procesar las opciones que indican.

Podemos empezar con algo como esto, que guardaremos en nuestra primera versión de `chkappconf`:

```
#!/bin/sh

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

Aquí utilizamos la estructura de control `while` del shell, que ejecuta el comando indicado como condición y, mientras dicho comando termine con éxito, ejecuta las sentencias entr `do` y `done`. Como condición,

```
test $# -gt 0
```

compara el número de argumentos con `0` usando *greater than* como comparación. Por último, dentro del cuerpo del `while` escribimos el primer argumento y posteriormente lo tiramos a la basura. La primitiva `shift` tira el primer argumento y conserva el resto.

Vamos a probarlo:

```
unix$ chkappconf -a b c
-a
b
c
unix$
```

Ahora estaría bien detener el while si el primer argumento no compienza por un "`-`". Por el momento podríamos usar `egrep` para eso. Hagamos un par de pruebas:

(c) 2016 Francisco J. Ballesteros

```
unix$ echo -a | egrep '^-'
-a
unix$ echo a | egrep '^-'
unix$
```

¡Estupendo! Pero sólo nos interesa ver si `egrep` dice que la entrada tiene ese aspecto o no. En lugar de enviar su salida a `/dev/null`, usaremos la opción `-q` (*quiet*) de `egrep` que hace que no imprima nada y tan sólo llame a *exit(2)* con el estatus adecuado.

```
unix$ echo -a | egrep -q '^-'
unix$ echo $?
0
unix$ echo a | egrep -q '^-'
unix$ echo $?
1
```

Ya casi estamos. Ahora necesitamos poder escribir como condición que tanto `test` compruebe que hay algún argumento como el comando anterior vea que comienza por un "`-`". Pero esto es fácil. El shell dispone de "`&&`" y "`||`" como operadores. Significan lo mismo que al evaluar condiciones en C, pero naturalmente esto es shell y utilizamos líneas de comandos. Nuestro script queda así por el momento:

```
#!/bin/sh

while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    echo option arg: $1
    shift
done
echo argv is $*
```

Y si lo ejecutamos, vemos que funciona:

```
unix$ chkappconf -a -ab c d
option arg: -a
option arg: -ab
argv is c d
unix$
```

El siguiente paso es procesar cada uno de los argumentos correspondientes a opciones. Aunque podríamos recurrir a `egrep` de nuevo, vamos a utilizar la estructura de control `case` del shell, que sabe como comprobar si un string encaja con una o más expresiones. Veamos antes un ejemplo:

```
unix$ x=ab
unix$ case $x in
> a*)
> echo a;;
> b*)
> echo b;;
> esac
a
unix$
```

Esta estructura compara el string entre `case` e `in` con cada una de las expresiones de cada rama del `case`. Estas expresiones son similares a las utilizadas para globbing y terminan en un "`)`". Tras cada expresión se incluyen las líneas de comandos que hay que ejecutar en dicho caso, terminando con un "`;;`". Por último,

(c) 2016 Francisco J. Ballesteros

se cierra el `case` con "`esac`".

Podemos escribir expresiones como

```
case ... in
[ab]c|d)
    ...
    ;;
esac
```

Esto es, es posible escribir conjuntos de caracteres "`[...]`" y también indicar expresiones alternativas separadas por un "`|`". Dado que `case` intentar encajar las expresiones en el orden en que se dan, y que "`*`" encaja con cualquier cosa, no existe `else` o `default` para `case`. Basta usar

```
case ... in
...
*)
    ... este es el default ...
    ;;
esac
```

Volvamos a nuestro script. Vamos a definir un par de variables para los flags y a comprobar si aparecen como opciones.

```
#!/bin/sh

dflag=y
sflag=y
while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    case $1 in
    *d*)
        dflag=y
        sflag=n
        ;;
    esac
    case $1 in
    *s*)
        sflag=y
        dflag=n
        ;;
    esac
    shift
done
echo dflag $dflag
echo sflag $sflag
echo argv is $*
```

Habitualmente habríamos usando "`n`" como valor inicial para los flags y los habríamos puesto a "`y`" si aparecen. Pero en este caso hay que procesar tanto directorios como shells si ninguno de los flags aparece. ¡Probémoslo!

(c) 2016 Francisco J. Ballesteros

```
unix$ chkappconf -d c d
dflag y
sflag n
argv is c d
unix$
```

Ya disponemos de `$dflag` para ver si hay que procesar sólo directorios y de `$sflag` para ver si hay que procesar sólo shells. Además, hemos dejado `$*` con el resto de argumentos.

Pero... ¿Y si hay una opción inválida? Bueno, siempre podemos recurrir a `echo` y `egrep` para ver si las opciones tienen buen aspecto:

```
...
while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    if echo $1 | egrep -q '[^-sd]' ; then
        echo usage: $0 '[-sd] [file...]' 1>&2
        exit 1
    fi
    case $1 in
    ...
done
...
```

Ahora podemos ponernos a hacer el trabajo según las opciones y los argumentos. Lo primero que haremos es dejar en una variable la lista de ficheros que hay que procesar.

```
files=$*
if test $# -eq 0 ; then
    files="/home/*/.myapp"
fi
```

Suponiendo que los ficheros de configuración se llaman `.myapp` y que queremos procesar dichos ficheros para todos los usuarios si no se indica ningún fichero.

Ahora podríamos procesar un fichero tras otro...

```
for f in $files ; do
    echo checking $f...
    if test $dflag = y ; then
        checkdirs $f
    fi
    if test $sflag = y ; then
        checkshells $f
    fi
done
```

Aquí vamos a utilizar `checkdirs` y `checkshells` para comprobar cada fichero. Aunque podríamos hacer otros scripts, parece que lo más adecuado es definir funciones. Y sí, ¡el shell permite definir funciones!.

Para definir una función escribimos algo como

(c) 2016 Francisco J. Ballesteros

```
myfun() {
    ...
}
```

En este caso, `myfun` es el nombre de la función. Tras el nombre han de ir los paréntesis (¡Sin nada entre ellos!) y después los comandos que constituyen el cuerpo entre llaves. Recuerda que estás usando el shell, esto no es C.

Dentro de la función, los argumentos se procesan como en un script. Así pues, en nuestro caso "`$1`" es el nombre del fichero que hay que comprobar y las funciones podrían ser algo como

```
checkdirs()
{
    echo checking dirs in $1
}
checkshells()
{
    echo checking shells in $1
}
```

Hemos terminado. Mostramos ahora el script entero para evitar confusiones.

```
#!/bin/sh

checkdirs()
{
    file=$1
    dirs=‘egrep ’^[a-z]+dir ’ config | \
        awk ’{print $1}’ | sort | uniq‘
    for d in $dirs ; do
        if test ! -d $d ; then
            echo no dir $d in $file
        fi
    done
}


checkshells()
{
    file=$1
    shells=‘egrep ’^shell ’ config | \
        awk ’{print $1}’ | sort | uniq‘
    for s in $shells ; do
        if test ! -x $x ; then
            echo no shell $s in $file
        fi
    done
}
```

(c) 2016 Francisco J. Ballesteros

```
dflag=y
sflag=y
while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    if echo $1 | egrep -q '[^-sd]' ; then
        echo usage: $0 '[-sd] [file...]' 1>&2
        exit 1
    fi
    case $1 in
    *d*)
        dflag=y
        sflag=n
        ;;
    esac
    case $1 in
    *s*)
        sflag=y
        dflag=n
        ;;
    esac
    shift
done


files=$*
if test $# -eq 0 ; then
    files="a b c"
fi
for f in $files ; do
    if test $dflag = y ; then
        checkdirs $f
    fi
    if test $sflag = y ; then
        checkshells $f
    fi
done
```

Hay formas mejores de hacer lo que hemos hecho. Pero lo que importa es que hemos podido hacerlo con las herramientas que tenemos. Por ejemplo, la práctica totalidad de los shells modernos incluyen *getopts(1)* para ayudar a procesar opciones y, además, tienes *getopt(1)* en cualquier caso para la misma tarea. Sólo con eso, ya podemos simplificar en gran medida nuestro script.

6. Editando streams

Ya hemos utilizado *sed(1)* para escribir las primeras líneas de un fichero. Pero puede hacer mucho más. Se trata de un editor similar a *ed(1)* (que fue el editor estándar en UNIX durante mucho mucho, pero hace ya mucho). Sus comandos están también disponibles (en general) en *vi(1)*, un editor "*visual*" que puede utilizarse en terminales de texto que no sean gráficos. La diferencia radica en que `sed` está hecho para procesar su entrada estándar, principalmente. De ahí el nombre *stream ed*.

`Sed` trabaja procesando su entrada línea a línea. En general, para cada línea, aplica los comandos de edición que se indican y después se escribe la línea en la salida.

Un comando en `sed` es una o dos *direcciones* seguidas de una función y, quizá, argumentos para la

(c) 2016 Francisco J. Ballesteros

función. Las direcciones determinan en qué líneas son aplicables los comandos.

Por ejemplo,

```
sed 5q
```

Significa "*en la línea 5*" ejecuta "*quit*". Dado que no se han ejecutado comandos que alteren las líneas, `sed` las escribe tal cual hasta llegar a la línea 5, momento en que termina. Por ello este comando escribe las 5 primeras líneas. La primera línea es la `1` para `sed`.

Sabiendo esto, podemos utilizar `sed` para escribir un rango de líneas. Por ejemplo

```
unix$ seq 15 | sed -n 5,7p
5
6
7
unix$
```

Aquí el flag `-n` hace que `sed` no imprima las líneas tras editarlas y utilizamos el comando `p` para imprimir las líneas entre la `5` y la `7`. Como puedes ver, dos direcciones separadas por una "`,`" delimitan un rango de líneas.

Veamos otro ejemplo, sabemos que `ps` escribe una primera línea con la cabecera que indica qué es cada campo. Por ejemplo:

```
unix$ ps
  PID TTY           TIME CMD
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
...
```

Podemos utilizar

```
unix$ ps | tail +1
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
...
```

como ya vimos o bien

```
unix$ ps | sed -n '2,$p'
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
...
```

Como puedes ver, "`$`" equivale al número de la última línea cuando se utiliza como dirección en `sed`.

De un modo similar podemos eliminar un rango de lineas con el comando `d`:

```
unix$ seq 15 | sed 2,12d
1
13
14
15
unix$
```

¡O varios rangos! El flag `-e` permite utilizar como argumento un comando de edición, y podemos repetirlo el número de veces que haga falta. Nos permite aplicar más de un comando en el mismo `sed`.

```
unix$ seq 15 | sed -e 1,3d -e 5,12d
4
13
14
15
unix$
```

Pero recuerda... ¡sed trabaja línea a línea sobre un stream!

```
unix$ seq 15 | sed -e 2,12d -e 1,3d
unix$
```

¡Aquí la hemos liado parda! Este comando no tiene mucho sentido dado que hay varios comandos que se aplican a líneas del mismo rango.

Las direcciones pueden ser expresiones regulares escritas entre "/". Esto nos permite seleccionar líneas entre ciertas líneas de interés. Por ejemplo, suponiendo que tenemos un fichero fuente en C con la función run, este comando escribe la cabecera y el cuerpo de dicha función:

```
unix$ sed -n  '/^run/,/^}/p <run2.c
run(char *cmd, char *argv[])
{
    ...
}
unix$
```

Para entender por qué no se escriben todas las líneas hasta la última que contenga un "}" al principio, piensa cómo evalúa sed las direcciones:

- Primero esperará hasta tener una línea que encaja con "^run"
- Una vez encontrada, seguirá hasta una línea que encaja con "^}".

En cada una de esas líneas ejecutará "p" (que imprime la línea) y una vez alcanza la línea de la segunda dirección, el comando termina por lo que no se vuelve a ejecutar para las líneas que siguen.

Otro comando de los más utilizados es el de sustituir una expresión por otra. Por ejemplo, este comando aumenta la tabulación del texto a que se aplica:

```
sed 's/^/ /'
```

Lo que hace es reemplazar el principio de línea por un tabulador (que hemos escrito como " ".

Supongamos que queremos renombrar todos los ficheros ".c" a ficheros ".C" por alguna extraña razón. Podemos utilizar sed para ello:

```
unix$ echo foo.c | sed 's/\.c$/.C/'
foo.C
```

Así pues

```
unix$ for f in *.c ; do
>    nf=`echo $f | sed 's/\.c$/.C/'`
>    mv $f $nf
>    done
unix$
```

hace el trabajo. Cuidado aquí. El comando de sed está entre comillas simples, pero todo lo que hay tras el signo "=" está entre comillas invertidas. ¡No son iguales!

Por ejemplo, podemos tener ficheros llamados ch1.w, ch2.w, etc. y llegar al ch10.w. En ese momento

quizá queramos renombrar `ch1.w` para que sea `ch01.w`, y así hasta el `9`. ¿Por qué? Bueno, de ese modo `ls` los lista en el orden adecuado y otros programas los verán en el orden en que deberían estar. De otro modo `ch11.w` estará listado antes que `ch2.w`.

Una vez más `sed` nos puede ayudar. Para probarlo, vamos a crear esos ficheros, y vamos a utilizar `seq` para crearlos.

```
unix$ for n in ‘seq 12‘; do touch ch$n.w ; done
unix$ ls
ch1.w    ch11.w    ch2.w    ch4.w    ch6.w    ch8.w
ch10.w    ch12.w    ch3.w    ch5.w    ch7.w    ch9.w
unix$
```

Y ahora podemos probar...

```
unix$ for ch in ch[0-9].w ; do
>    echo mv $ch ‘echo $ch | sed ’s/\([0-9]\)/0\1/’ ‘
>    done
mv ch1.w ch01.w
mv ch2.w ch02.w
mv ch3.w ch03.w
mv ch4.w ch04.w
mv ch5.w ch05.w
mv ch6.w ch06.w
mv ch7.w ch07.w
mv ch8.w ch08.w
mv ch9.w ch09.w
unix$
```

En lugar de ejecutar `mv` directamente, lo hemos precedido de `echo` para ver lo que va a ejecutar. Si nos convence el resultado podemos añadir "`|sh`" para ejecutarlo o quitar el `echo`.

Pero la parte interesante es "`s/\([0-9]\)/0\1/`". Esto quiere decir que hay que sustituir el texto que encaja con "`[0-9]`" en cada línea por "`0\1`". El "`\1`" significa *el texto que encaja en la primera expresión entre* "`\(`...`\)`", que en este caso es "`[0-9]`". Así pues, "`\1`" es capaz de recuperar parte del texto que ha encajado en la expresión para utilizarlo al reemplazar. Vuelve a leer la sesión anterior y notarás el efecto de "`\1`".

Observa que esta facilidad te permite reordenar el texto que hay en la línea. Si contamos de 10 a 15 de 1 en 1 usando `seq`

```
unix$ seq 10 1 15
10
11
12
13
14
15
```

podemos ver el efecto de estas expresiones en otro ejemplo:

(c) 2016 Francisco J. Ballesteros

```
unix$ seq 10 1 15 | sed 's/\([0-9]\)\([0-9]\)/\2\1 and not \1\2/'
01 and not 10
11 and not 11
21 and not 12
31 and not 13
41 and not 14
51 and not 15
unix$
```

Ahora tenemos dos subexpresiones entre "`\(...\)`", y reemplazamos toda la expresión por texto consistente en los dos caracteres con el orden cambiado, luego "` and not `" y luego los dos caracteres en el orden original. Ni que decir tiene que puedes utilizar cualquier subexpresión y no una que opere sobre un sólo carácter.

Para borrar una expresión, podemos reemplazarla por *nada*. Por ejemplo, este comando elimina un nivel de tabulación de su entrada:

```
sed 's/^ //'
```

Debes tener cuidado con las sustituciones en `sed`. Se aplican al primer trozo de la línea que encaja con la expresión, pero no se repiten más adelante en la misma línea. Esto es, puedes reemplazar texto una sóla vez. Si deseas reemplazar todas las veces que sea posible puedes añadir un flag "`g`" al comando. Por ejemplo, dados los ficheros `chXX.w`", este comando nos imprime los números para esos ficheros:

```
unix$ ls
ch1.w    ch11.w    ch2.w    ch4.w    ch6.w    ch8.w
ch10.w    ch12.w    ch3.w    ch5.w    ch7.w    ch9.w
unix$ ls | sed 's/[.a-z]//g'
    1
    10
    11
    12
    2
    ...
    9
unix$
```

Compara con

```
unix$ ls | sed 's/[.a-z]//'
h1.w
h10.w
...
unix$
```

7. Trabajando con tablas

En gran cantidad de casos tendrás datos tabulados como en una hoja de cálculo. Gran número de ficheros de configuración en UNIX presentan este aspecto, y además la salida de muchos comandos está tabulada.

La mayoría de las veces puedes operar filtrando filas y columnas y trabajar con los comandos que hemos visto durante el curso. Para cálculos numéricos puedes utilizar cualquier de las calculadores de línea de comandos, aunque tal vez sea más simple utilizar *expr(1)* para evaluar expresiones numéricas simples, como en:

```
unix$ n=4
unix$ n=‘expr $n ’*’ 2‘
unix$ echo $n
8
unix$
```

No necesitas mucho más.

No obstante, dispones de *awk(1)* que es un lenguaje pensado para operar sobre datos tabulados. Ya lo hemos utilizado para seleccionar campos, pero ahora vamos a ver algunos otros usos típicos y como son los programas de `awk` en general.

La idea es similar a *sed(1)*. Para `awk` los ficheros constan de registros (líneas) que tienen campos (separados por blancos, aunque ya sabes como cambiar esto). Lo que hace `awk` es leer la entrada (o el fichero indicado como argumento) línea a línea y aplicar el programa sobre el mismo.

Un programa en `awk` se suele escribir directamente en el primer argumento en la llamada a `awk`, pero ya sabes que puedes utilizar "`#!`" en scripts para lo que gustes. El programa consta de **reglas**, que tienen una expresión (opcional) que selecciona a qué lineas se aplica la regla y una acción entre llaves (opcional) que indica qué debe hacer `awk` en esas líneas. Si no hay expresión se considera que la acción se aplica a todas las líneas. Si no hay acción se considera que la acción es imprimir.

En las expresiones y en las acciones tienes predefinidas las variables "`$1`" (el primer campo), "`$2`" (el segundo), etc. El registro entero, la línea, es "`$0`". Y además tienes definidas las variables `NR` *number of record* (número de línea) y `NF` *number of field* (número de campos).

Para jugar, vamos a utilizar el fichero `/etc/passwd` que define las cuentas de usuario. Tiene este aspecto:

```
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
...
```

El nombre de usuario, seguido de su uid, gid, nombre real, home y por último el shell que utiliza. Todos los campos separados por "`:`".

Vamos a imprimir tan sólo el nombre de usuario y el shell que utilizan:

```
unix$ awk -F: ’{print $1, $NF}’ /etc/passwd
nobody /usr/bin/false
root /bin/sh
daemon /usr/bin/false
...
```

En lugar de print, podemos utilizar `printf`. Las acciones de `awk` utilizan sintáxis casi como la de C y las variables no es preciso declararlas. Sabiendo esto puedes ser optimista y hacer intentos, la página de manual te sacará de dudas en cualquier caso.

```
unix$ awk -F: ’{printf("user %s shell %s\n", $1, $NF);}’ /etc/passwd
user nobody shell /usr/bin/false
user root shell /bin/sh
user daemon shell /usr/bin/false
...
```

¡Ahora sólo para *root*!

```
unix$ awk -F: '
>    $1 == "root" {
>        printf("user %s shell %s\n", $1, $NF);
>    }' /etc/passwd
user root shell /bin/sh
unix$
```

En este caso la expresión `$1 == "root"` hace que la acción ejecute sólo si el primer campo es el string `root`.

Pero compara con este otro comando:

```
unix$ awk -F: '
>    $1 ~ /root/ {
>        printf("user %s shell %s\n", $1, $NF);
>    }' /etc/passwd
user root shell /bin/sh
user _cvmsroot shell /usr/bin/false
unix$
```

El operador "`~`" permite ver si cierto texto encaja con una expresión regular (escrita entre "`/`"). Por eso aparecen dos cuentas, porque el nombre de usuario contiene `root` en ambas.

Igual que en el caso de `sed`, podemos utilizar como expresión un par de expresiones regulares separadas por una coma, lo que hace que la expresión sea cierta para las líneas comprendidas entre una que encaja con la primera y otra que encaja con la segunda.

```
unix$ seq 15 | awk '/3/,/5/ {print}'
3
4
5
13
14
15
unix$
```

Verás que aparecen las líneas entre la 3 y la 5 (que encajan con las expresiones) y las líneas entre la 13 y la 15 (que también lo hacen). Las expresiones pueden ser mas elaboradas y puedes utilizar los operadores "`&&`" y "`||`" casi como en C.

La acción `next` salta el registro en curso. Considerando que `awk` procesa las reglas en el orden en que las escribimos, podemos hacer que imprima un rango de líneas pero salte algunas. Por ejemplo,

```
unix$ seq 15 | awk '
>    $0 == "4" {next}
>    /3/,/5/ {print} '
3
5
13
14
15
unix$
```

Muy útil para utilizar

```
$0 ~ /^#/ {next}
```

al principio del programa de `awk` e ignorar las líneas que comienzan por un "#" (que suele ser el carácter

(c) 2016 Francisco J. Ballesteros

de comentario en el shell y en muchos ficheros).

Las expresiones BEGIN y END hacen que su acción ejecute antes de procesar la entrada y después de haberla procesado. Podemos utilizar esto para sumar los números de la entrada, por ejemplo:

```
unix$ seq 15 | awk '
>    {tot += $0}
>    END {print tot}'
120
unix$
```

O para obtener la media:

```
unix$ seq 15 | awk '
>    {tot += $0}
>    END {print tot/NR}'
8
unix$
```

Este otro obtiene la media de los números pares, y además los imprime:

```
unix$ seq 15 | awk '
>    $0 % 2 == 0 {
>        print;
>        tot += $0;
>    }
>    END {print tot/NR}'
2
4
6
8
10
12
14
3.73333
```

Pero podríamos haberlo hecho así:

```
unix$ seq 15 | awk '
>    { if ($0 %2 == 0) {
>        print;
>        tot += $0;
>      }
>    }'
2
4
...
```

Simplemente recuerda que las acciones disponen de sintaxis similar a la de C y se optimista. Consulta *awk(1)* para ver qué funciones tienes disponibles y que expresiones y estructuras de control. Si has entendido lo que hemos estado haciendo, seguro que te sobra con la página de manual.

¿Puedes escribir un comando que numere las líneas de un fichero? Para imprimirlo con números de línea, por ejemplo.

(c) 2016 Francisco J. Ballesteros

Capítulo 7: Concurrencia

(c) 2016 Francisco J. Ballesteros

1. Threads y procesos

Hemos utilizado `fork` para crear procesos, lo que hace que los procesos hijo *no* compartan recursos con el padre: tienen su propia copia de los segmentos de memoria, descriptores de fichero, etc. Esa es la abstracción, aunque en realidad, el segmento de texto se suele compartir (dado que es de sólo lectura). El resto de segmentos se comportan como si no se compartiesen, pero UNIX hace que el padre y el hijo los compartan tras degradar sus permisos a sólo-lectura. Cuando un proceso intenta escribir en una de las páginas que "no comparten", UNIX comprueba que, en efecto, el proceso puede escribir en ella y hace una copia de la página que comparten padre e hijo. Como en este punto cada proceso tiene su propia copia de la página, los permisos vuelven a dejarse como lectura-escritura, y el proceso puede completar su escritura pensando que siempre ha tenido su propia copia. A esto se lo conoce como **copy on write**.

Un proceso para el kernel es, principalmente, el flujo de control (su juego de registros ya sea en el procesador o salvado en su pila de kernel y su pila, tanto la de usuario como la de kernel). El resto de recursos puede o no compartirlos con el proceso que lo ha creado.

A la vista de esto, resulta posible crear un nuevo proceso que comparta la memoria con el padre. A esto se lo suele denominar **thread**. El nombre procede de los tiempos en que el kernel no sabía nada de threads y éstos eran *procesos ligeros de usuario* o *corutinas* creadas por la librería de C (u otra similar) sin ayuda del kernel. Desde hace años, los threads son procesos como cualquier otro y el kernel los planifica como a cualquier otro proceso. Lo único que sucede es que algunos procesos comparten segmentos de memoria (y otros recursos) con otros procesos. Eso es todo. Si piensas en los threads como en procesos todo irá bien.

Resulta más cómodo (y es más portable) crear un proceso que comparte los recursos con el padre utilizando la *librería de pthreads* que está instalada en prácticamente todos los sistemas UNIX que utilizando la llamada al sistema involucrada (que suele además variar mucho de unos UNIX a otros).

El siguiente programa crea tres threads que imprimen 5 mensajes cada uno.

[thr.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <string.h>


static void*
tmain(void *a)
{
    int i;
    char *arg, *sts;

    arg = a;
    for(i = 0; i < 5; i++) {
        write(2, arg, strlen(arg));
    }
    asprintf(&sts, "end %s", strchr(arg, 't'));
    free(arg);
    return sts;
}
```

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    int i;
    pthread_t thr[3];
    void *sts[3];
    char tabs[10], *a;

    for(i = 0; i < 3; i++) {
        memset(tabs, '\t', sizeof tabs);
        tabs[i] = 0;
        asprintf(&a, "%st %d\n", tabs, i);
        if(pthread_create(thr+i, NULL, tmain, a) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], sts+i);
        printf("join %d: %s\n", i, sts[i]);
        free(sts[i]);
    }
    exit(0);
}
```

El programa crea un nuevo thread utilizando código como

```
pthread_t thr;
...
if (pthread_create(&thr, NULL, func, funcarg) != 0) {
    err(1, "thread");
}
```

La llamada crea un proceso que comparte los recursos con el que hace la llamada y arregla las cosas para que el nuevo flujo de control ejecute `func(funcarg)`. Donde `func` es el *programa principal* o el punto de entrada del nuevo thread y ha de tener una cabecera como

```
void *func(void *arg)
```

El argumento que pasamos al final a `pthread_create` es el argumento con que se llamará a dicho punto de entrada. Además, la función principal del thread devuelve un `void*` que hace las veces de *exit status* para el thread.

El primer argumento es un puntero a un **tid** o identificador de thread que utiliza la librería de threads para identificar al thread en cuestión. Utilizarás este valor en sucesivas llamadas que se refieran al thread que has creado. Es similar al *pid* de un proceso. Piensa que aunque cada thread tiene un proceso, la librería mantiene más información sobre cada thread y desea utilizar sus propios identificadores.

El programa puede después llamar a

```
void *sts;
...
pthread_join(thr, &sts);
```

para (1) esperar a que el thread termine y (2) obtener su estatus. Dicho de otro modo, `pthread_join` hace las veces de *wait(2)*.

(c) 2016 Francisco J. Ballesteros

Cuando no deseamos llamar a `pthread_join` para un thread, debemos informar a la librería *pthread* de tal cosa (¡Igual que sucedía con `fork` y `wait`!). Una buena forma de hacerlo es hacer que la función principal del thread llame a

```
pthread_t me;
me = pthread_self();
pthread_detach(me);
```

Pero observa que este código ejecuta en el nuevo thread, no en el código del proceso padre. La función `pthread_self` es como *getpid(2)*, pero devuelve el *thread id* y no el *process id*.

Si ahora vuelves a leer el programa seguramente entiendas las partes que antes te parecían oscuras. Cuando lo ejecutamos, podemos ver una salida parecida a esta:

```
unix$ thr
        t 2
        t 2
        t 2
        t 2
    t 1
t 0
    t 1
t 0
t 0
t 0
t 0
    t 1
    t 1
    t 1
        t 2
join 0: end t 0
join 1: end t 1
join 2: end t 2
unix$
```

Siendo procesos distintos... ¡No sabemos en qué orden van a ejecutar! Luego la salida seguramente difiera si repetimos la ejecución. Pero podemos ver que los tres threads ejecutan concurrentemente que que el programa principal puede esperar correctamente a que terminen y recuperar el estatus de salida de cada uno de ellos.

Seguramente resultará instructivo que ahora intentes leer de nuevo el programa trazando mentalmente cómo ha podido producir la salida que hemos visto.

2. Condiciones de carrera

¿De nuevo?... ¡Sí! En cuanto más de un proceso utiliza el mismo recurso... hay condiciones de carrera. Ahora que podemos compartir memoria entre varios procesos vamos a verlo de nuevo.

El siguiente programa incrementa un contador un número dado de veces (10 por omisión) en tres threads distintos:

(c) 2016 Francisco J. Ballesteros

[race.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <string.h>

enum { Nloops = 10 };
static int nloops = Nloops;


static void*
tmain(void *a)
{
    int i;
    int *cntp;

    cntp = a;
    for(i = 0; i < nloops; i++) {
        *cntp = *cntp + 1;
    }
    return NULL;
}


int
main(int argc, char* argv[])
{
    int i, cnt;
    pthread_t thr[3];
    void *sts[3];

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    cnt = 0;
    for(i = 0; i < 3; i++) {
        if(pthread_create(thr+i, NULL, tmain, &cnt) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], sts+i);
        free(sts[i]);
    }
    printf("cnt is %d\n", cnt);
    exit(0);
}
```

Si lo ejecutamos, el contador debiera ser tres veces el número de incrementos, ¿No?. Y parece que es así...

(c) 2016 Francisco J. Ballesteros

```
unix$ thr
cnt is 30
unix$
```

Pero... ¡Vamos a ejecutarlo para que cada thread haga 1000 incrementos!

```
unix$ thr 1000
cnt is 2302
unix$
```

¡Otra vez!

```
unix$ thr 1000
cnt is 2801
unix$
```

No te gustaría que pasara esto si se tratase del programa que controla ingresos en tu cuenta corriente. Estamos viendo simplemente el efecto de una condición de carrera en el uso de la variable `cnt`, que es compartida por todos los procesos del programa (el proceso que teníamos desde el programa principal y los tres threads que hemos creado).

Podemos verlo fácilmente si simplificamos el programa para que ejecute sólo dos threads y para que la función que ejecutan sea:

[race2]:

```
...
static int cnt;
static void*
tmain(void *a)
{
    int i;
    for (i =0 ; i < 2; i++) {
        cnt++;
    }
    printf("cnt is %d\n", cnt);
    return NULL;
}
...
```

Ahora haremos dos incrementos en dos threads y hemos cambiado la declaración de `cnt` para que sea una variable global, por simplificar más.

Cuando lo ejecutamos vemos:

```
unix$ race2
cnt is 2
cnt is 4
```

Todo bien.

Cambiemos otra vez el código para que sea:

[race3]:
```
static void*
tmain(void *a)
{
    int i, loc;
    for (i =0 ; i < 2; i++) {
        loc = cnt;
        loc++;
        cnt = loc;
    }
    printf("cnt is %d\n", cnt);
    return NULL;
}
```

Si lo ejecutamos, vemos que todo sigue bien:

```
unix$ race3
cnt is 2
cnt is 4
```

Pero si hacemos el siguiente cambio:

[race4]:
```
static void*
tmain(void *a)
{
    int i, loc;
    for (i =0 ; i < 2; i++) {
        loc = cnt;
        loc++;
        sleep(1);
        cnt = loc;
    }
    printf("cnt is %d\n", cnt);
    return NULL;
}
```

¡La cosa cambia!

```
unix$ race4
cnt is 2
cnt is 2
```

Ambos threads escriben 2 como valor final para `cnt`. Hemos provocado que la condición de carrera se manifieste. Esto quiere decir que, aunque no seamos conscientes, todas las versiones anteriores de este programa están mal y no pueden utilizarse.

El problema de la condición de carrera procede en realidad de la *ilusión* implementada por los procesos: *ejecución secuencial independiente*. Resulta que si tenemos un sólo procesador, la ejecución no es ni secuencial ni independiente. UNIX multiplexa (reparte) el procesador entre los procesos, y aun así pensamos que nuestros programas ejecutan secuencialmente y sin interferencias.

Lo que sucede en realidad es que las instrucciones de los procesos se *mezclan* en un único flujo de control implementado por el procesador. Esto es, ejecutará determinado número de instrucciones de un proceso, luego tendremos un cambio de contexto y ejecutará otro, luego otro, etc. No sabemos cuándo sucederán los cambios de contexto y por tanto no sabemos en qué orden se mezclaran las instrucciones. Se suele llamar

(c) 2016 Francisco J. Ballesteros

interleaving (entrelazado) al mezclado de instrucciones, por cierto.

Las primeras veces que hemos utilizado el programa resulta que todas las instrucciones involucradas en

```
cnt++
```

han ejecutado. Cuando hemos cambiado el código para que sea más parecido a las instrucciones que realmente ejecutan

```
loc = cnt;
loc++;
cnt = loc;
```

hemos seguido teniendo (mala) suerte y dichas instrucciones han ejecutado sin interrupción.

Así pues, la ilusión de ejecución secuencial y sin interferencia se ha mantenido. Cuando incrementamos un registro para incrementar la variable el mundo seguía como lo dejamos en la línea anterior y la variable global (en la memoria) seguía teniendo el mismo valor. Cuando actualizamos en la siguiente línea la variable global nadie había consultado ni cambiado la variable mientras ejecutamos.

Al introducir la llamada a `sleep` hemos provocado un cambio de contexto justo en el punto en que tenemos la condición de carrera (durante la consulta e incremento de la global). El efecto puede verse en la figura 19.

El problema consiste en que, después de haber consultado el valor del contador en la local `loc` y antes de que actualicemos el valor del contador, *otro* proceso accede al contador y puede que incluso lo cambie. En la figura podemos ver que el entrelazado ha sido:

1. El thread 1 consulta la variable
2. El thread 2 consulta la variable
3. El thread 1 incrementa su copia de la variable (registro o variable local)
4. El thread 1 incrementa su copia de la variable (registro o variable local)
5. El thread 1 actualiza la variable
6. El thread 2 actualiza la variable

Este *interleaving* pierde un incremento. El problema es que toda la secuencia de código utilizando la variable no ejecuta de forma indivisible (se "cuela" otro proceso que utiliza la misma variable). Si este código fuese **atómico**, esto es, ejecutase de forma indivisible, no habría condición de carrera; pero no lo es.

Por ello es crítico que no se utilice la variable global desde ningún otro proceso mientras la consultamos, incrementamos y actualizamos y llamamos **región crítica** a dicho fragmento de código. Una *región crítica* es simplemente código que accede a un recurso compartido y que plantea condiciones de carrera si no las evitamos haciendo que ejecute de forma atómica (indivisible con respecto a otros que comparten el recurso).

¿Qué sucedía en el programa inicial que hacía *n* incrementos en 3 threads? Simplemente que hay cierta probabilidad de tener un cambio de contexto en la región crítica. La probabilidad aumenta cuantas más veces ejecutemos la región crítica. Al ejecutar mil veces el incremento en cada thread, *algunos* de los incrementos sufrieron un cambio de contexto en mal sitio, eso es todo.

3. Cierres

¿Cómo podemos evitar una condición de carrera como la que hemos visto? La respuesta viene de la definición del problema. Necesitamos que la región crítica ejecute atómicamente (de forma indivisible en lo que se refiere a otros que utilizan el mismo recurso). Decimos que necesitamos **exclusión mutua**. Esto es,

(c) 2016 Francisco J. Ballesteros

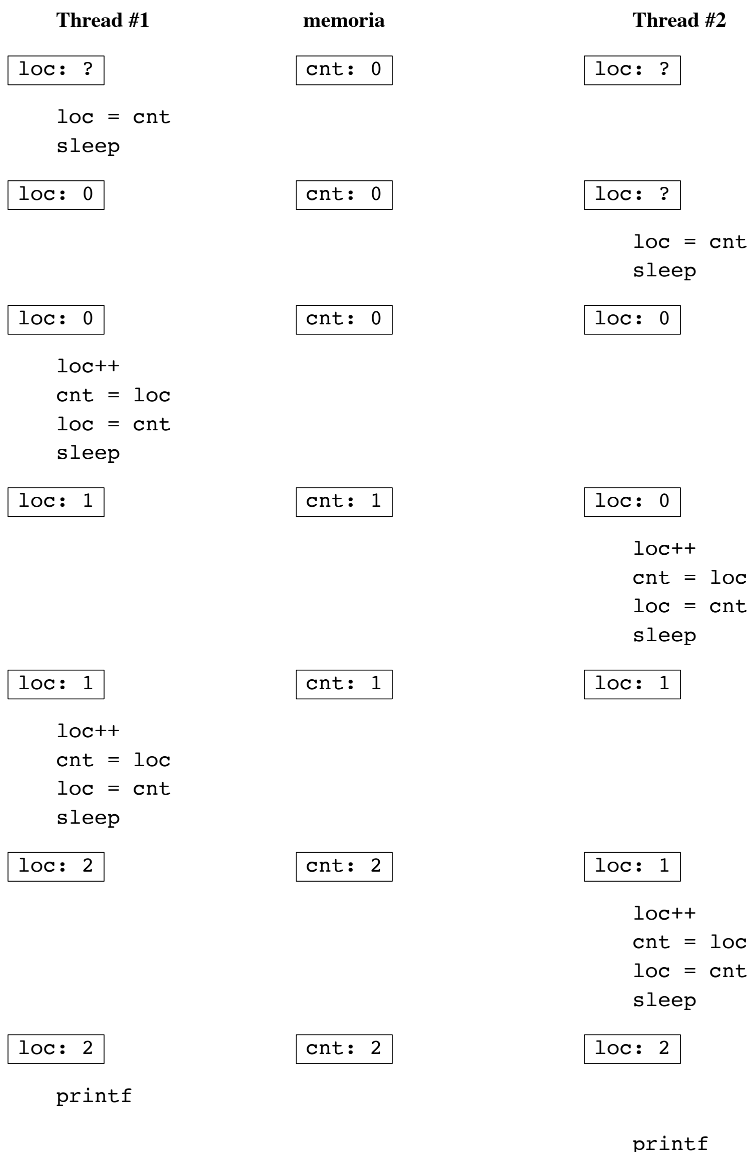

Figura 19: *Un entrelazado de sentencias en ambos procesos que da lugar a una condición de carrera en la última versión del programa*.

que si un proceso está en la región crítica, otros no puedan estarlo. Dicho de otro modo, que si un proceso está utilizando el recurso compartido otros no puedan hacerlo.

En la figura 20 puede verse lo que sucede si `cnt++` ejecuta de forma atómica. En este caso, al contrario que antes, uno de los procesos ejecuta `cnt++` antes que otro. El que llega después puede ejecutar `cnt++` partiendo del valor resultante del primero, sin que exista problema alguno. Incluso si no sabemos en qué orden ejecutan los `cnt++`, el valor final resulta correcto.

Dependiendo del recurso al que accedamos, tenemos diversas abstracciones para conseguir exclusión mutua en la región crítica. Lo que necesitamos es una abstracción como mecanismo de **sincronización** para que unos procesos se pongan de acuerdo con otros.

En el caso de procesos que comparten memoria podemos utilizar una abstracción llamada **cierre** para conseguir tener exclusión mutua. La idea es rodear la región crítica por código similar a...

(c) 2016 Francisco J. Ballesteros

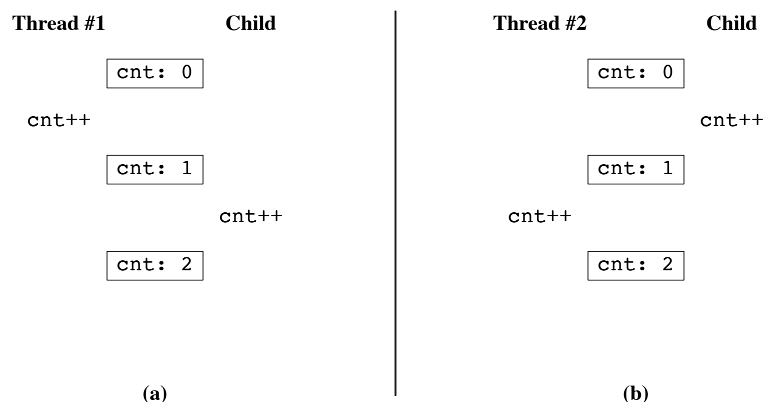

Figura 20: *Incrementando el contador de forma atómica no hay condiciones de carrera, ya sea (a) o (b) lo que suceda en realidad.*

```
lock(cierre);
... región crítica...
unlock(cierre);
```

La variable `cierre` y sus dos operaciones se comportan como una "llave" (de ahí el nombre). Sólo un proceso puede tener el cierre. Cuando un proceso llama a `lock`, si el cierre está abierto (libre), el proceso echa el cierre y continúa. Si el cierre está echado (ocupado) el proceso espera (dentro de `lock`) y, cuando esté libre, el proceso echa el cierre y continúa. La operación `unlock` suelta o libera el cierre.

Existen instrucciones capaces de consultar y actualizar una posición de memoria de forma atómica y pueden utilizarse para implementar `lock` y `unlock`. Basta usar un entero y suponer que si es cero el cierre está libre y si no lo es está ocupado.

Dependiendo del tipo de cierre que utilicemos es posible que el proceso esté en un `while` esperando a que el cierre se libere. En tal caso el proceso utiliza el procesador para esperar y decimos que tenemos **espera activa**. A estos cierres se los conoce como **spin locks**. Igualmente, es posible que los cierres cooperen con UNIX y que el sistema pueda bloquear al proceso mientras espera. Esto último es lo deseable, pero has de consultar el manual para ver qué cierres tienes disponibles en tu librería de threads.

Cuando un cierre (u otra abstracción) se utiliza para dar exclusión mutua se lo denomina **mutex**. Suelen usarse las expresiones *coger el mutex* y *soltar el mutex* para indicar que se echa el cierre y se libera.

Cuando utilizamos *pthreads* tenemos a nuestra disposición el tipo de datos `pthread_mutex_t` que representa un mutex y funciones para adquirir y liberar el mutex. Para implementar exclusión mutua basta con usar

```
pthread_mutex_lock(&lock);
...región crítica...
pthread_mutex_unlock(&lock);
```

Donde la variable `lock` puede declararse e inicializarse según

```
pthread_mutex_t lock;
...
pthread_mutex_init(&lock, NULL);
```

Una vez deje de ser útil el mutex, hay que liberar los recursos que usa llamando a

```
pthread_mutex_destroy(&lock);
```

(c) 2016 Francisco J. Ballesteros

Un aviso. Todas estas llamadas devuelven en realidad una indicación de error. Normalmente 0 is hacen el trabajo y algún otro número si no. Hemos optado por no comprobar estos errores en los programas que mostramos en esta sección para no distraer del código que requiere la programación concurrente. Pero **hay que comprobar los errores** cuando se utilicen en la práctica. Muchas veces sólo fallan si el proceso se queda sin memoria disponible, pero eso no es una excusa para no comprobar errores.

El siguiente programa es similar al del epígrafe anterior pero no presenta condiciones de carrera.

[safe.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <string.h>

typedef struct Cnt Cnt;
struct Cnt {
    int n;
    pthread_mutex_t lock;
};

enum { Nloops = 10 };
static int nloops = Nloops;


static void*
tmain(void *a)
{
    int i;
    Cnt *cntp;

    cntp = a;
    for(i = 0; i < nloops; i++) {
        pthread_mutex_lock(&cntp->lock);
        cntp->n = cntp->n + 1;
        pthread_mutex_unlock(&cntp->lock);
    }
    return NULL;
}
```

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    int i;
    Cnt cnt;
    pthread_t thr[3];
    void *sts[3];

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    cnt.n = 0;
    if(pthread_mutex_init(&cnt.lock, NULL) != 0) {
        err(1, "mutex");
    }
    for(i = 0; i < 3; i++) {
        if(pthread_create(thr+i, NULL, tmain, &cnt) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], sts+i);
        free(sts[i]);
    }
    pthread_mutex_destroy(&cnt.lock);
    printf("cnt is %d\n", cnt.n);
    exit(0);
}
```

Hemos seguido la costumbre de situar el cierre cerca del recurso al que cierra, si ello es posible. En este caso, situamos tanto el cierre como el contador dentro de la misma estructura:

```
typedef struct Cnt Cnt;
struct Cnt {
    int n;
    pthread_mutex_t lock;
};
```

Cuando los threads intentan incrementar el contador, ejecutan

```
pthread_mutex_lock(&cntp->lock);
cntp->n = cntp->n + 1;
pthread_mutex_unlock(&cntp->lock);
```

y uno de ellos llegará a `pthread_mutex_lock` antes que los demás. Ese proceso adquiere el cierre y continúa. Si llegan otros a `pthread_mutex_lock` antes de que el proceso que tiene el mutex lo suelte, quedarán bloqueados dentro de `pthread_mutex_lock` hasta que el cierre quede libre. Si ahora editamos el código y añadimos un `sleep` a mitad del incremento, como hicimos antes, veremos que no tenemos condiciones de carrera. Podemos ver esto además en la siguiente ejecución:

```
unix$ safe 10000
cnt is 30000
unix$
```

¡No se pierden incrementos!

(c) 2016 Francisco J. Ballesteros

Por cierto, los mutex de *pthread(3)* suelen bloquear el proceso para hacer que espere, por lo que no son *spin locks* y no desperdician procesador.

Otro detalle importante es que es posible inicializar un cierre usando un valor inicial, en lugar de usar `pthread_mutex_init`. Para hacerlo, basta usar código como:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

4. Cierres en ficheros

El siguiente programa incrementa un contador que tenemos escrito dentro de un fichero. Esto sucede en la realidad en aplicaciones que deben numerar secuencialmente recursos cada vez que ejecuta determinado programa, por ejemplo. Si tenemos un fichero `datafile` que contiene "3" y ejecutamos el programa, el fichero pasará a contener 4. El código del programa es simple:

[incr.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("datafile", O_RDWR);
    if(fd < 0) {
        err(1, "open");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);

    nb++;
    fprintf(stderr, "set to %d\n", nb);
    snprintf(buf, sizeof buf, "%d", nb);
    lseek(fd, 0, 0);
    if(write(fd, buf, strlen(buf)) != strlen(buf)) {
        err(1, "write");
    }
    close(fd);
    exit(0);
}
```

Hemos incluído multiples prints que no tendríamos normalmente para que veamos qué sucede al ejecutarlo. Vamos a hacerlo:

```
unix$ cat datafile
3
unix$ incr
nb is 3
set to 4
unix$ cat datafile
4
unix$
```

Todo bien.

Pero pongamos un `sleep` de tal forma que el código use ahora

```
nb++;
sleep(5);
```

en lugar de tan sólo incrementar `nb`. Y ahora ejecutemos dos veces el programa:

```
unix$ incr &
[1] 47846
nb is 4
unix$ incr
nb is 4
set to 5
set to 5
unix$ cat datafile
5
unix$
```

¡Hemos perdido un incremento!

Naturalmente, dos procesos que acceden al mismo fichero producen una condición de carrera por compartir el fichero. Necesitamos un cierre. Si ambos procesos compartiesen memoria podríamos utilizar un cierre como los que hemos antes sin ningún problema (¡Aunque el recurso que cierran sea un fichero y esté fuera de la memoria del proceso!). Recuerda que todo esto es un convenio. Hemos quedado en adquirir un cierre antes de entrar en la región crítica, pero es tan sólo un acuerdo.

En este caso, por desgracia, los procesos no comparten memoria. Pero aún podemos solucionar el problema con la ayuda de UNIX. En UNIX es posible adquirir un cierre sobre un fichero e incluso sobre un rango de bytes dentro de un fichero. La llamada al sistema para conseguirlo es *flock(2)*. Concretamente,

```
flock(fd, LOCK_EX);
```

echa el cierre en `fd` de modo exclusivo (como en los cierres que vimos antes) y

```
flock(fd, LOCK_UN);
```

libera el cierre. Si el proceso muere o cierra el descriptor de fichero, el cierre se libera.

Sabiendo esto, el siguiente programa es la versión correcta del programa anterior, sin condiciones de carrera.

(c) 2016 Francisco J. Ballesteros

[safeincr]:
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("datafile", O_RDWR);
    if(fd < 0) {
        err(1, "open");
    }
    if(flock(fd, LOCK_EX) != 0){
        err(1, "lock");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);

    nb++;

    fprintf(stderr, "set to %d\n", nb);
    snprintf(buf, sizeof buf, "%d", nb);
    lseek(fd, 0, 0);
    if(write(fd, buf, strlen(buf)) != strlen(buf)) {
        err(1, "write");
    }
    if(flock(fd, LOCK_UN) != 0){
        err(1, "lock");
    }
    close(fd);
    exit(0);
}
```

Recuerda que todo esto es un convenio. ¡Podríamos echar el cierre en un fichero para trabajar en otro! Así pues, tenemos ya la forma de trabajar con ficheros compartidos. Basta pensar dónde y cómo disponemos ficheros que usamos como cierre. Por ejemplo, muchos programas de correo utilizan un fichero llamado `.LOCK` en el directorio que contiene los buzones de correo (que son ficheros). Para utilizar los ficheros en dicho directorio, estos programas llaman a `flock` sobre `.LOCK` y luego trabajan con los buzones. Cuando terminan de trabajar, sueltan el cierre. No es muy diferente a lo que hicimos nosotros utilizando una variable `lock` para tener exclusión mutua en el acceso a un contador `cnt`.

(c) 2016 Francisco J. Ballesteros

5. Cierres de lectura/escritura

Como en ocasiones nos preguntamos qué valor tendrá el contador que incrementamos en el programa anterior, vamos a realizar un programa para imprimirlo. Podríamos utilizar *cat(1)*, naturalmente, pero vamos a hacer un programa que pueda ver el valor sin condiciones de carrera.

[safecat]:
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("datafile", O_RDONLY);
    if(fd < 0) {
        err(1, "open");
    }
    if(flock(fd, LOCK_EX) != 0){
        err(1, "lock");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);
    if(flock(fd, LOCK_UN) != 0){
        err(1, "lock");
    }
    close(fd);
    exit(0);
}
```

Podemos usarlo sin condiciones de carrera:

```
unix$ safecat
nb is 5
unix$
```

¿Qué sucede si ejecutamos tres `safecat` y un `safeinr` simultáneamente? Todos ellos adquieren el cierre sobre el fichero para trabajar en él con exclusión mutua y, aunque el orden en que consigan ejecutar y echar el cierre variará, podríamos tener una ejecución como la que vemos en la figura 21.

Si hay muchos procesos leyendo (ejecutando `safecat`) tardaremos mucho en ejecutarlo todo. Pero hay una posibilidad de mejorar las cosas: puesto que los lectores sólo leen el fichero, es posible ejecutar más de un lector a la vez sin exclusión mutua respecto a otros lectores. Dicho de otro modo, podríamos tener un

(c) 2016 Francisco J. Ballesteros

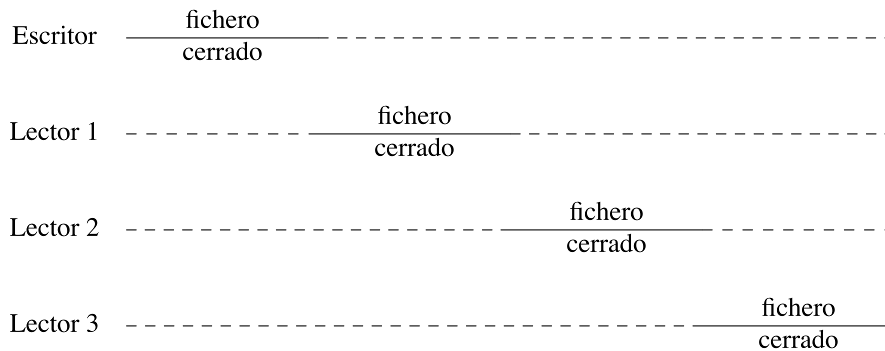

Figura 21: *Múltiples lectores y escritores de un fichero con un cierre exclusivo. Sólo puede ejecutar uno cada vez dentro de la región crítica.*

escritor o cualquier número de lectores, pero no ambas cosas. Existe un tipo de cierre que permite exclusión mutua entre lectores y escritores. Permite lectores concurrentes en exclusión con escritores. Naturalmente, los escritores excluyen otros escritores también. Si utilizamos este tipo de cierre, la ejecución podría ser como se ven la figura 22. Como puede verse, los procesos han de esperar en menos y, conjuntamente, terminamos antes.

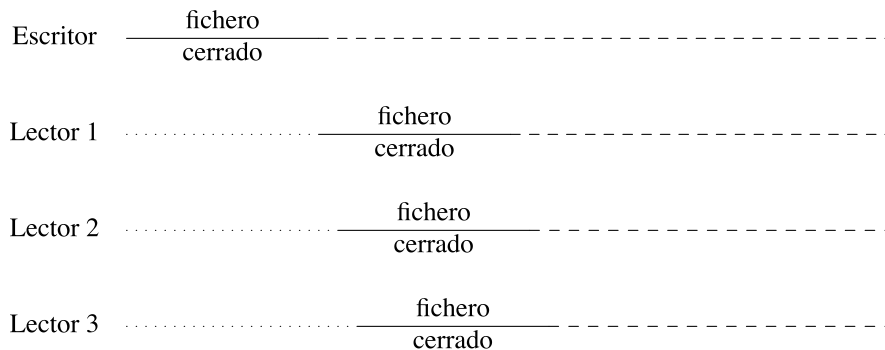

Figura 22: *Con un cierre de tipo lectores/escritores permitimos múltiples lectores concurrentes manteniendo la exclusión mutua de los escritores.*

Otra forma de verlo (que en realidad coincide con la implementación) es pensar que los escritores comparten el mutex con otros lectores cuando lo adquieren.

En nuestro programa podemos conseguir este efecto haciendo que `safecat` adquiera el cierre en modo *lector*. Si hacemos tal cosa, estamos utilizando el cierre del fichero como un cierre de tipo lectores/escritores, también llamado **read/write lock**.

[safecatr]:
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("afile", O_RDONLY);
    if(fd < 0) {
        err(1, "open");
    }
    if(flock(fd, LOCK_SH) != 0){
        err(1, "lock");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);
    if(flock(fd, LOCK_UN) != 0){
        err(1, "lock");
    }
    close(fd);
    exit(0);
}
```

Ahora tenemos algunos (lectores) que adquieren un cierre usando `LOCK_SH` (*shared*) y otros (escritores) que lo hacen usando `LOCK_EX` (*exclusive*).

6. Cierres de lectura/escritura para threads

La librería de *pthreads* dispone de cierres de tipo lectura/escritura. Su uso es similar al que hemos visto para usar los mutex de *pthread*, pero las variables de tipo cierre se declaran e inicializan como en

```
pthread_rwlock_t rwlock;
...
pthread_rwlock_init(&rwlock);
```

o bien

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Un lector usaría estas llamadas para proteger su región crítica:

```
pthread_rwlock_rdlock(&rwlock);
    ... región crítica ...
pthread_rwlock_unlock(&rwlock);
```

Y un escritor estas otras:

```
pthread_rwlock_wrlock(&rwlock);
    ... región crítica ...
pthread_rwlock_unlock(&rwlock);
```

Una vez deja de ser necesario el cierre hay que liberar sus recursos con

```
pthread_rwlock_destroy(&rwlock);
```

Si un proceso necesita un cierre como lector y posteriormente convertirse en escritor, suele ser mejor mantener el cierre como escritor todo el tiempo.

7. Deadlocks

Rara vez necesitaremos un único cierre. O, dicho de otro modo, rara vez necesitaremos un único recurso compartido. Lo normal es compartir diversos recursos. En principio cada recurso puede utilizar un cierre distinto y los procesos que necesitan acceder al recurso pueden adquirir su cierre durante la región crítica. Pero piensa lo que sucede si un proceso ejecuta

```
lock(a);
lock(b);
...
```

y otro en cambio ejecuta

```
lock(b);
lock(a);
```

Si ejecutamos el primer `lock` de cada proceso, ninguno podrá continuar jamás. Cada uno tiene un cierre y necesita el que tiene el otro. El efecto neto es que los procesos se quedan bloqueados de por vida y no terminan. A esto lo denominamos **deadlock** o **interbloqueo**. ¿Recuerdas qué sucedía si un proceso deja abierto un descriptor de escritura en un pipe y se pone a leer del mismo? Efectivamente, es un deadlock.

En el caso de cierres, para evitar el problema suele bastar fijar un **orden** a la hora de adquirir los recursos. Si todos adquieren los cierres que necesitan siguiendo el orden acordado, no tenemos interbloqueos. Por ejemplo, podemos acordar que el cierre de `a` ha de tomarse antes que el de `b` y el de `b` antes que el de `c`. De ser así, podríamos tener código para un proceso que ejecute

```
lock(a);
lock(c);
...
```

otro que ejecute

```
lock(a);
lock(b);
```

y otro

```
lock(b);
lock(c);
```

y en ningún caso tendríamos un bloqueo. Si alguien necesita un cierre nadie puede tenerlo y además estar esperando los que ya tenemos nosotros: no hay deadlocks.

(c) 2016 Francisco J. Ballesteros

Esto puede resultar complicado y en ocasiones se opta por usar un único cierre para todos los recursos, a lo que se suele denominar *giant lock* o "cierre gordo". Pero naturalmente se limita la concurrencia y se reduce el rendimiento. No obstante, hay menos posibilidades de tener condiciones de carrera por olvidar echar un cierre en el momento adecuado.

Hay veces en que se opta por dejar que suceda el deadlock y, tras detectarlo o comprobar que estamos ante un posible deadlock, soltar todos los cierres y volverlo a intentar. Esto no es realmente una solución dado que podríamos volver a caer en el interbloqueo una vez tras otra (aunque no es 100% seguro que siempre suceda tal cosa). A este tipo de bloqueos (que podrían romperse con cierta probabilidad) se los denomina **livelock**.

Lo que es más, puede que algún proceso tenga mala suerte y, si aplica esta estrategia, nunca consiga los cierres que necesita porque siempre gane otro proceso al obtener uno de los cierres que necesita. El pobre proceso continuaría sin poder trabajar, a lo que se denomina **starvation**, o **hambruna**.

8. Semáforos

Quizá la abstracción más conocida para sincronizar procesos en programación concurrente y controlar el acceso a los recursos sean los **semáforos**. Un semáforo es en realidad un contador que indica cuántos *tickets* tenemos disponibles para acceder a determinado recurso. No son muy diferentes de un contador de entradas para acceder al cine. Si no hay entradas, no se puede entrar al cine. Del mismo modo, si un semáforo está a `0`, no se puede acceder al recurso de que se trate.

La parte interesante del semáforo es que dispone de dos operaciones:

```
down(sem);
```

y

```
up(sem);
```

que (respectivamente) adquieren un *ticket* y lo liberan.

- La primera, `down`, toma un ticket del semáforo (cuyo valor se decrementa) cuando existen tickets (el valor es positivo). De no haber tickets disponibles (cuando el semáforo es cero), `down` *espera* a que existan tickets libres y entonces toma uno.
- La llamada `up` simplemente libera un ticket. Si alguien estaba esperando en un `down`, lo adquiere y continúa. Si nadie estaba esperando en un `down`, el ticket queda en el semáforo, que pasa a incrementarse.

A la abstracción se la denomina semáforo puesto que su inventor pensó en semáforos ferroviarios que controlan el acceso a las vías. Por cierto que en muchas implementaciones se utilizan los nombres

```
P(sem);
```

y

```
V(sem);
```

en lugar de `down` y `up`, y en otras implementaciones se utiliza

```
wait(sem);
```

y

```
signal(sem);
```

en su lugar. Nosotros usaremos `down` y `up` para evitar confusiones con otras operaciones.

Dado un semáforo, podemos implementar un mutex utilizando código como

(c) 2016 Francisco J. Ballesteros

```
Sem sem = 1;
...
down(sem);
...región crítica...
up(sem);
```

Puesto que sólo hay un ticket en el semáforo, sólo un proceso puede adquirirlo, con lo que tenemos exclusión mutua.

9. Semáforos en UNIX

Existen diversas implementaciones de semáforos en UNIX. Normalmente tienes disponible una denominada *posix semaphores*, que puedes combinar con los threads de la librería de *pthread(3)*.

Así pues... ¡Cuidado! Es posible que si usas determinado tipo de semáforos estos sólo existan en tu sistema y no en otros UNIX. Por ejemplo, Linux dispone de semáforos con nombre y sin nombre, pero OS X suministra semáforos con nombre y no dispone del otro tipo de semáforos. Nosotros usaremos sólo semáforos con nombre que tenemos disponibles en gran parte de los sistemas UNIX hoy en día.

Veamos el siguiente programa:

[semcnt.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <err.h>
#include <string.h>

static int cnt;
static sem_t *sem;

enum { Nloops = 10 };
static int nloops = Nloops;


static void*
tmain(void *a)
{
    int i;

    for(i = 0; i < nloops; i++) {
        if (sem_wait(sem) < 0) {
            err(1, "down");
        }
        cnt++;
        if (sem_post(sem) < 0) {
            err(1, "up");
        }
    }
    return NULL;
}
```

```
int
main(int argc, char* argv[])
{
    int i;
    pthread_t thr[3];
    void *sts;
    char name[1024];

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    snprintf(name, sizeof name, "/sem.%s.%d", argv[0], getpid());
    sem = sem_open(name, O_CREAT, 0644, 1);
    if (sem == NULL) {
        err(1, "sem creat");
    }
    printf("sem '%s' created\n", name);
    for(i = 0; i < 3; i++) {
        if(pthread_create(thr+i, NULL, tmain, &cnt) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], &sts);
    }
    if (sem_close(sem) < 0) {
        warn("sem close");
    }
    sem_unlink(name);

    printf("cnt is %d\n", cnt);
    exit(0);
}
```

Se trata una vez más de nuestro programa para incrementar un contador en tres threads el número indicado de veces, y podemos comprobar que funciona sin condiciones de carrera:

```
unix$ semcnt 10000
sem '/sem.semcnt.49029' created
cnt is 30000
unix$
```

Lo primero que hemos de tener en cuenta es que estos semáforos *tienen nombre* y existen como una abstracción que suministra el kernel. UNIX mantendrá un record dentro con la implementación del semáforo. Eso quiere decir que hemos de destruirlo cuando no sean útil.

Puesto que tienen nombre, hemos de tener cuidado de no utilizar nombres que usen otros programas. En nuestro caso optamos por construir un nombre a partir de "`sem`" (el nombre de nuestra variable), `argv[0]` (el nombre de nuestro programa) y `getpid()`. Por ejemplo,

```
/sem.semcnt.49029
```

es el nombre del semáforo que hemos creado en la ejecución anterior. Haciéndolo así, es imposible que colisionemos con otros nombres de semáforo de nuestro programa o de otros.

El semáforo lo ha creado la llamada

```
sem = sem_open(name, O_CREAT, 0644, 1);
if (sem == NULL) {
    err(1, "sem creat");
}
```

que es similar a *open(2)* cuando se usa para crear fichero. Esta vez creamos un semáforo y no un fichero. Tras los permisos se indica *qué número de tickets* queremos inicialmente en el semáforo. En este caso `1` dado que es un mutex.

Cuando todo termine, necesitamos cerrar y destruir el semáforo utilizando

```
if (sem_close(sem) < 0) {
    warn("sem close");
}
sem_unlink(name);
```

Una vez creado, podemos compartir el semáforo con otros procesos que no compartan memoria. Basta abrir el semáforo en ellos utilizando

```
sem = sem_open(semname, 0);
if (sem == NULL) {
    // no existe!
}
```

después de haberlo creado. No obstante, es mejor utilizar en ellos la misma llamada que usamos en nuestro programa, indicando `O_CREAT` para que el semáforo se cree si no existe.

10. ¿Y si algo falla?

Un problema con este tipo de semáforos es que siguen existiendo hasta que se llame a `sem_unlink` para borrarlos. Si nuestro programa falla, el semáforo seguirá existiendo en el kernel hasta que rearranquemos o detengamos el sistema. Lamentablemente, no disponemos de comando alguno para listar los semáforos que existen y es posible que dejemos semáforos perdidos si nuestro programa falla.

Un remedio paliativo es usar como nombre del semáforo uno que no incluya el *pid* del proceso y tan sólo use el nombre de nuestra aplicación y escribir un programa que borre el semáforo (o borrarlo antes de crearlo por si habíamos dejado alguno en ejecuciones anteriores.

Otro problema importante es que si un proceso muere mientras tiene un ticket el ticket se pierde.

En conclusión, si se desea un mutex es mucho mejor utilizar un cierre en un fichero si hemos de sincronizar procesos distintos que no forman parte del mismo programa.

Este problema afecta a muchos otros semáforos y abstracciones disponibles en UNIX. Presta atención a qué hace el sistema ante una muere prematura de un proceso que tiene un mutex. Lo deseable es que el mutex se libere, pero posiblemente no suceda tal cosa. Los cierres en ficheros con `flock` si que se comportan correctamente y es por ello que son populares a la hora de conseguir un mutex para compartir recursos entre procesos totalmente distintos.

Cuando los procesos forman parte del mismo programa basta con que los mutex que creamos se liberen si el programa muere. Si uno de los procesos del programa muere prematuramente, esto se debe a un bug y dado que es el mismo programa no afecta a otras aplicaciones, por lo que no es un problema en realidad: habrá que depurar el error y listo.

11. Semáforos con pipes

Vamos a implementar un semáforo que podremos usar en aplicaciones que compartan un proceso padre (o ancestro) común. La idea es utilizar un pipe como semáforo y guardar tantos bytes en el semáforo como tickets queramos tener: Para hacer un `down` leeremos y byte y para hacer un `up` escribiremos un byte.

Con esta idea, el único problema es que es preciso crear el semáforo antes de llamar a `fork` (para que todos lo procesos lo compartan), si es que usamos `fork`. Si usamos threads, dado que comparten los descriptores de fichero, no tenemos problemas.

Este es el interfaz para nuestros semáforos

[sem.h]:

```
typedef struct Sem Sem;
struct Sem {
    int fd[2];
};

int semdown(Sem *s);
int semup(Sem *s);
int semcreat(Sem *s, int val);
void semclose(Sem *s);
```

y esta es la implementación:

[sem.c]:

```
#include <stdio.h>
#include <unistd.h>
#include "sem.h"

int
semdown(Sem *s)
{
    char c;

    if (read(s->fd[0], &c, 1) != 1) {
        return -1;
    }
    return 0;
}


int
semup(Sem *s)
{
    if (write(s->fd[1], " ", 1) != 1) {
        return -1;
    }
    return 0;
}
```

(c) 2016 Francisco J. Ballesteros

```
void
semclose(Sem *s)
{
    close(s->fd[0]);
    close(s->fd[1]);
    s->fd[0] = s->fd[1] = -1;
}


int
semcreat(Sem *s, int n)
{
    int i;

    if (pipe(s->fd) < 0) {
        return -1;
    }
    for (i = 0; i < n; i++) {
        if (semup(s) < 0) {
            semclose(s);
            return -1;
        }
    }
    return 0;
}
```

Una vez los tenemos, podemos utilizarlos como en nuestro programa de ejemplo, que hemos adaptado para utilizar estos semáforos:

[semcnt.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <string.h>
#include "sem.h"

enum { Nloops = 10 };
static int nloops = Nloops;
static int cnt;
static Sem sem;
```

```
static void*
tmain(void *a)
{
    int i;

    for(i = 0; i < nloops; i++) {
        if (semdown(&sem) < 0) {
            err(1, "down");
        }
        cnt++;
        if (semup(&sem) < 0) {
            err(1, "up");
        }
    }
    return NULL;
}


int
main(int argc, char* argv[])
{
    int i;
    pthread_t thr[3];
    void *sts;

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    if (semcreat(&sem, 1) < 0) {
        err(1, "sem creat");
    }
    for(i = 0; i < 3; i++) {
        if(pthread_create(thr+i, NULL, tmain, &cnt) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], &sts);
    }
    semclose(&sem);
    printf("cnt is %d\n", cnt);
    exit(0);
}
```

Ahora podemos compilarlo y ejecutarlo sin problemas:

```
unix$ cc -c sem.c
unix$ cc -c semcnt.c
unix$ cc -o semcnt semcnt.o sem.o
unix$ semcnt 10000
cnt is 30000
unix$
```

(c) 2016 Francisco J. Ballesteros

Un problema con estos semáforos es que no podemos hacer que el número de tickets supere el número de bytes que caben en el buffer del pipe. ¿Qué pasaría si lo hacemos? Una ventaja es que cuando nuestro programa termine su ejecución se cerrarán sus descriptores de fichero incluso si hemos olvidado llamar a `semclose` y el semáforo (el pipe) se destruirá sin dejar recursos perdidos en el sistema. Otra ventaja es que podemos usarlos tanto si creamos procesos que llamen a `fork` como si usamos `pthread_create`.

12. Buffers compartidos: el productor/consumidor

Un problema que aparece a todas horas en programación concurrente es el del **productor-consumidor**, también conocido como el del **buffer acotado**. Se trata de tener procesos que producen cosas y procesos que las consumen, como puede verse en la figura 23. Cuando el buffer no tiene límite se denomina *productor-consumidor* al problema y en otro caso se le conoce como el problema del *buffer acotado*. ¡Los pipes son un caso de dicho problema!

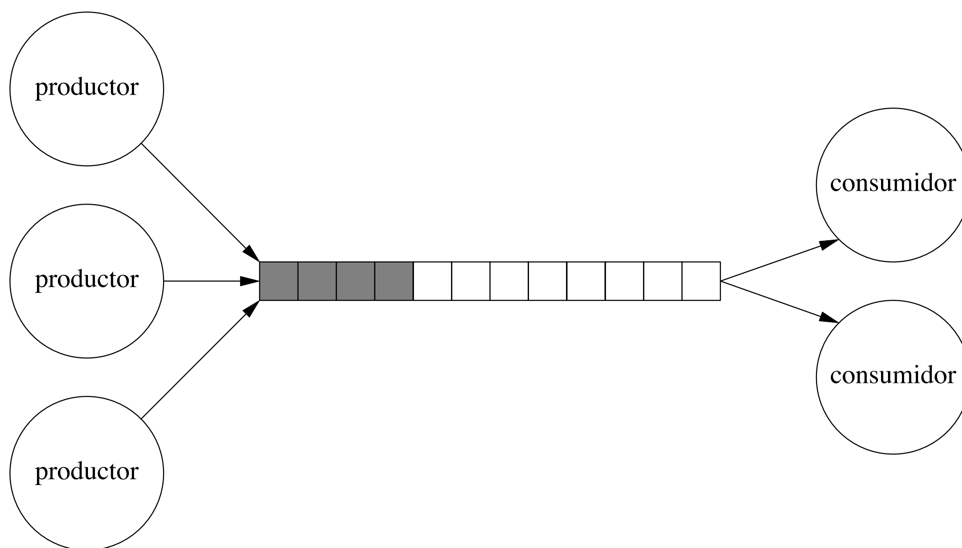

Figura 23: *El problema del productor consumidor con un buffer acotado.*

La solución es sencilla si pensamos en lo siguiente:

- Para producir un ítem es preciso tener un hueco en el buffer
- Para consumir in ítem es preciso tener un ítem en el buffer
- Para operar con el buffer es preciso utilizar un mutex.

La idea es tener un semáforo que represente los ítems en el buffer, otro que represente los huecos en el buffer y otro que represente el mutex:

- Cuando queramos un hueco basta pedirlo: `down(huecos)`.
- Cuando queramos un ítem basta pedirlo: `down(ítems)`.

El mutex ya sabes manejarlo. Naturalmente, si alguien produce un ítem deberá llamar a `up(ítems)` e, igualmente, si alguien produce un hueco deberá llamar a `up(huecos)`. Si piensas en los semáforos como en cajas con tickets, todo esto te resultará natural.

Primero vamos a implementar un buffer compartido que, en nuestro caso, será una cola de caracteres. Vamos a mostrar y discutir el código en el mismo orden en que aparece en el fichero `pc.c`, secuencialmente de arriba a abajo.

(c) 2016 Francisco J. Ballesteros

[pc.c]:
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <err.h>
#include "sem.h"

enum {QSIZE = 4};

typedef struct Queue Queue;
struct Queue {
    Sem mutex, nchars, nholes;
    char buf[QSIZE];
    int hd, tl;
};
```

En la cola `Queue` guardamos los tres semáforos que hemos mencionado, el buffer para guardar los caracteres y la posición de la cabeza (`hd`) y cola (`tl`) en el buffer. Insertaremos caracteres al final de la cola (en `tl`) y los tomaremos del principio, de `hd`.

Inicializar la cola requiere crear los semáforos y poco más:

```
static int
qinit(Queue *q)
{
    memset(q, 0, sizeof *q);
    if (semcreat(&q->mutex, 1) < 0) {
        return -1;
    }
    if (semcreat(&q->nchars, 0) < 0) {
        semclose(&q->mutex);
        return -1;
    }
    if (semcreat(&q->nholes, QSIZE) < 0) {
        semclose(&q->mutex);
        semclose(&q->nchars);
        return -1;
    }
    return 0;
}
```

Tomamos la precaución de dejar todos los campos a valores nulos llamando a `memset` para que todo esté bien inicializado.

Terminar de usar la cola requiere cerrar nuestros semáforos:

```
static void
qterm(Queue *q)
{
    semclose(&q->mutex);
    semclose(&q->nchars);
    semclose(&q->nholes);
    q->hd = q->tl = 0;
}
```

¡Ahora hay que hacer la parte interesante! Para poner un carácter en la cola necesitamos y hueco, luego ponerlo mientras tenemos el mutex (para evitar condiciones de carrera en el uso de la cola) y además hemos de echar un ticket al semáforo que indica cuántos ítems hay en la cola:

```
static int
qput(Queue *q, int c)
{
    if (semdown(&q->nholes) < 0) {
        return -1;
    }
    if (semdown(&q->mutex) < 0) {
        return -1;
    }
    q->buf[q->tl] = c;
    q->tl = (q->tl+1)%QSIZE;
    if (semup(&q->mutex) < 0) {
        return -1;
    }
    if (semup(&q->nchars) < 0) {
        return -1;
    }
    return 0;
}
```

Tomar un carácter de la cola es simétrico con respecto a ponerlo. Esta vez pedimos ítems y generamos huecos:

```
static int
qget(Queue *q)
{
    int c;

    if (semdown(&q->nchars) < 0) {
        return -1;
    }
    if (semdown(&q->mutex) < 0) {
        return -1;
    }
    c = q->buf[q->hd];
    q->hd = (q->hd+1)%QSIZE;
    if (semup(&q->mutex) < 0) {
        return -1;
    }
    if (semup(&q->nholes) < 0) {
        return -1;
    }
    return c;
}
```

Ya podemos declarar la cola:

```
static Queue q;
```

Un *productor* será un proceso que pone caracteres en la cola. En nuestro caso cada productor pondrá 10:

```
static void*
tput(void *a)
{
    char *s;
    int i;

    s = a;
    for (i = 0; i < 10; i++) {
        if (qput(&q, *s) < 0) {
            err(1, "qput");
        }
    }
    return NULL;
}
```

Hemos pasado como parámetro un puntero a un `char` para que cada productor ponga un carácter distinto en la cola.

El consumidor va a sacar de la cola todo lo que pueda hasta que obtenga un `0`, con lo que marcaremos el final de los datos:

```
static void*
tget(void *a)
{
    int c;
    char buf[1];

    for (;;) {
        c = qget(&q);
        if (c == 0) {
            break;
        }
        if (c < 0) {
            err(1, "qget");
        }
        buf[0] = c;
        if (write(1, buf, 1) != 1) {
            err(1, "write");
        }
    }
    return NULL;
}
```

Por último, nuestro programa principal inicializará la cola y creará dos productores y un consumidor.

```
int
main(int argc, char* argv[])
{
    pthread_t p1, p2, g;
    void *sts;

    if (qinit(&q) < 0) {
        err(1, "qinit");
    }
    if (pthread_create(&p1, NULL, tput, "a") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&p2, NULL, tput, "b") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&g, NULL, tget, NULL) != 0) {
        err(1, "thread");
    }
    pthread_join(p1, &sts);
    pthread_join(p2, &sts);
    if (qput(&q, 0) < 0) {
        err(1, "qput");
    }
    pthread_join(g, &sts);
    write(1, "\n", 1);
    qterm(&q);
    exit(0);
}
```

¡Listos para ejecutarlo!

(c) 2016 Francisco J. Ballesteros

```
unix$ pc
bbaabaaaaababbabbabbb
unix$
```

Este problema es muy importante. En casi todos los programas utilizarás algún tipo de buffer compartido y necesitarás código que resuelva el problema del productor-consumidor. Además, es un buen problema para terminar de entender cómo se utilizan los semáforos y cómo funcionan. Observa que si el buffer se llena, el productor espera a tener hueco. Igualmente, si el buffer se vacía, el consumidor espera a tener algo que consumir. ¡Los pipes funcionan de este modo!

13. Monitores

Existe otra abstracción básica para conseguir la sincronización entre procesos a la hora de acceder a recursos compartidos. Se trata del **monitor**.

Un monitor es una abstracción que, en teoría, suministra el lenguaje de programación y presenta un aspecto similar a un módulo o paquete. La diferencia con respecto a un módulo (o paquete) radica en que se garantiza que *sólo un proceso puede ejecutar dentro del monitor en un momento dado*. Dicho de otro modo, tenemos exclusión mutua en el acceso al monitor.

Los datos compartidos se declararían dentro del monitor y sólo pueden ser utilizados llamando a operaciones del monitor.

Así pues, podríamos utilizar un contador compartido sin tener condiciones de carrera si escribimos algo como:

```
monitor sharedcnt;
static int cnt;
void
incr(int delta)
{
    cnt += delta;
}
int
get(void)
{
    return cnt;
}
```

La idea es que desde fuera del monitor, podemos utilizar llamadas del estilo a

```
sharedcnt c;
c.incr(+3);
printf("val is %d\n", c.get());
```

sin tener que preocuparlos por la programación concurrente.

Esta abstracción es tan simple de usar y suele entenderse tan bien que habitualmente siempre se programa pensando en ella. ¡Tengamos monitores o no! De hecho, rara vez tenemos monitores de verdad. Normalmente tenemos las herramientas para implementarlos y para programar pensando en ellos, y ese es el caso en UNIX.

Para implementar un monitor basta con declarar un mutex para el monitor y

- adquirir el mutex al principio de cada operación del monitor y
- soltar el mutex al final de cada operación del monitor.

Tan sencillo como eso.

Por ejemplo, para el caso del contador compartido podríamos implementar el monitor como una estructura que contiene el contador y el mutex del monitor: Una vez más, vamos a describir el código en el mismo orden en que aparece en el fichero en C del programa.

[cntmon.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <err.h>
#include "sem.h"

typedef struct Cnt Cnt;
struct Cnt {
    Sem mutex;
    int val;
};
static Cnt cnt;
```

En este caso, `cnt` es nuestro monitor. Pero necesitamos programar las operaciones del monitor. Primero, una para inicializarlo

```
static int
cntinit(Cnt *c)
{
    memset(c, 0, sizeof *c);
    if (semcreat(&c->mutex, 1) < 0) {
        return -1;
    }
    return 0;
}
```

y otra para terminar su operación

```
static void
cntterm(Cnt *c)
{
    semclose(&c->mutex);
}
```

Después, una función para cada operación del monitor teniendo cuidado de mantener el mutex del monitor cerrado durante toda la función.

(c) 2016 Francisco J. Ballesteros

```
static int
cntincr(Cnt *c, int delta)
{
    if (semdown(&c->mutex) < 0) {
        return -1;
    }
    c->val += delta;
    if (semup(&c->mutex) < 0) {
        return -1;
    }
    return 0;
}
[verb
static int
cntget(Cnt *c, int *valp)
{
    if (semdown(&c->mutex) < 0) {
        return -1;
    }
    *valp = c->val;
    if (semup(&c->mutex) < 0) {
        return -1;
    }
    return 0;
}
```

Y ya tenemos implementado el monitor. Ahora podemos declararlo:

```
static Cnt c;
```

Un thread que desee incrementar el contador utiliza la operación del monitor correspondiente para hacerlo:

```
static void*
tincr(void *a)
{
    if (cntincr(&c, 1) < 0) {
        err(1, "cntintr");
    }
    return NULL;
}
```

Nótese como podemos utilizar `cntintr` casi como en un programa secuencial, sin pensar mucho en la programación concurrente.

El programa principal va a inicializar el monitor, crear varios threads que incrementen el contador que contiene y por último imprimir cuánto vale dicho contador utilizando la operación `cntget` del monitor.

(c) 2016 Francisco J. Ballesteros

```
int
main(int argc, char* argv[])
{
    pthread_t thr[10];
    void *sts;
    int i, val;

    if (cntinit(&c) < 0) {
        err(1, "cntinit");
    }
    for (i = 0; i < 10; i++) {
        if (pthread_create(&thr[i], NULL, tincr, NULL) != 0) {
            err(1, "thread");
        }
    }
    for(i = 0; i < 10; i++) {
        pthread_join(thr[i], &sts);
    }
    if (cntget(&c, &val) < 0) {
        err(1, "cntget");
    }
    printf("val %d\n", val);
    cntterm(&c);
    exit(0);
}
```

Si ejecutamos el programa podemos utilizar el contador compartido sin condiciones de carrera:

```
unix$ cntmon
val 10
unix$
```

¿Podríamos entonces programar

```
if (cntget(&c, &val) < 0) {
    err(1, "cntget");
}
if (val > 0) {
    if (cntincr(&c, -1) < 0) {
        err(1, "cntincr");
    }
}
```

para decrementar un contador sólo si es positivo? En teoría el monitor nos permite solucionar las condiciones de carrera... ¿No? ¡Lo que no impide es que cometamos estupideces!

Pensemos. La llamada a `cntget` funciona correctamente, y deja en `val` el valor del contador. Igualmente, la llamada a `cntincr` incrementa el contador (en `-1` en este caso). Lo que ocurre es que entre una y otra llamadas podría ser que otro proceso entre y decremente el contador. Este es el problema de no mantener cerrado el recurso durante *toda la región crítica*.

En este ejemplo, todo el código debería formar parte de una operación del monitor: *decrementar-si-podemos*.

(c) 2016 Francisco J. Ballesteros

14. Variables condición

Vamos a implementar el problema del buffer acotado utilizando monitores. Al contrario que cuando utilizamos semáforos, aquí la idea es poder tener operaciones en el monitor que podamos llamar olvidando la concurrencia.

En principio tendríamos una operación para poner un ítem en el buffer y otra para consumirlo. Ambas utilizarían un buffer declarado *dentro* del monitor y ambas tendrían el cierre (o el mutex) del monitor mientras ejecutan. Luego no habría condiciones de carrera en el acceso al buffer.

El problema viene en cuanto vemos que para implementar `put` necesitamos esperar a tener un hueco para poder continuar. De tener monitores, nos gustaría poder escribir

```
void
put(int item)
{
    if(buffer lleno) {
        wait until(tenemos hueco)
    }
    buffer[nitems++] = item;
}
```

El código se entiende, ¿No? Dentro de `put` tenemos el mutex del monitor, no te preocupes por condiciones de carrera. Habría bastado

```
buffer[nitems++] = item;
```

si el buffer nunca se llena. El problema es que si el buffer puede llenarse hay que esperar a tener un hueco antes de consumirlo.

Igualmente, el consumidor podría utilizar una operación como la que sigue:

```
int
get(void)
{
    if(buffer vacio) {
        wait until(tenemos un item)
    }
    return buffer[--nitems];
}
```

Primero esperamos si es que el buffer está vacío a que deje de estarlo y luego consumimos uno de los elementos del buffer.

Para poder esperar hasta que se cumpla determinada condición que necesitamos *dentro de un monitor* tenemos **variables condición**. Son variables que representan una condición y tienen dos operaciones:

- `wait`: espera incondicionalmente a que se cumpla la condición
- `signal`: avisa de que la condición se cumple.

Nótese que estamos comprobando con un `if` si la condición se cumple o no antes de esperar a que se cumpla. Dicho de otro modo, `wait` en una variable condición duerme al proceso incondicionalmente. En cambio, `wait` en un semáforo (recuerda que era un posible nombre para `down`) sólo duerme al proceso si el semáforo está sin tickets.

Otra forma de verlo es que con los semáforos es el semáforo el que se ocupa de si tenemos que dormir o no. Nosotros tan sólo pedimos un ticket y cuando lo tengamos `down` (o `wait`) nos dejará continuar. Pero con las variables condición somos nosotros los que decidimos esperar cuando vemos que no podemos continuar.

(c) 2016 Francisco J. Ballesteros

Para que el código de nuestro problema esté completo falta llamar a `signal` cuando se cumplan las condiciones. Si le damos un repaso, este sería nuestro código por el momento:

```
Cond hayhuecos, hayitems;
void
put(int item)
{
    if(ntimes == SIZEOFBUFFER) {
        wait(hayhuecos);
    }
    buffer[nitems++] = item;
    signal(hayitems);
}


int
get(void)
{
    int item;
    if(nitems == 0) {
        wait(hayitems);
    }
    item = buffer[--nitems];
    signal(hayhuecos);
    return item;
}
```

Un buen nombre para una variable condición es el nombre de la condición. Cuando se llama a `wait`, el proceso se duerme soltando el mutex del monitor para que otros procesos puedan utilizar el monitor mientras dormimos. Cuando se llama a `signal`, uno de los procesos que duermen despierta. Si no hay procesos dormidos esperando, `signal` no hace nada.

Esto quiere decir que deberíamos llamar a `signal` justo al final de la operación, de tal modo que nosotros terminamos y el proceso que hemos despertado es el único que continúa ejecutando dentro del monitor.

Las variables condición forman parte de la implementación del monitor y se ocupan de soltar el mutex mientras duermen. Dependiendo de la implementación, lo normal es que cuando un proceso despierta en un `signal`, el que lo despierta le ceda el mutex.

En otras ocasiones (lamentablemente) el proceso que despierta compite por el mutex del monitor con todos los demás, lo que quiere decir que otro proceso podría ganarle y hacer que la condición de nuevo sea falta. Eso implica que el código en este caso debería ser

```
while(nitems == 0) {
    wait(hayitems);
}
```

y no

```
if(nitems == 0) {
    wait(hayitems);
}
```

puesto que cuando despertamos es posible que tengamos que volver a dormir si otro se nos adelante antes de que podamos adquirir el mutex del monitor. Java es un notable ejemplo de esta **pésima** implementación de monitores.

(c) 2016 Francisco J. Ballesteros

En UNIX disponemos de llamadas en la librería *pthread(3)* para usar variables condición. Dichas llamadas utilizan tanto una variable condición como un mutex que tenemos que crear para que lo use el monitor. Esto es, podemos implementar monitores pero los estamos programando casi a mano.

Sigue el código de nuestro productor consumidor utilizando los mutex y variables condición de *pthread(3)*, en lugar de utilizar nuestros semáforos como en la implementación que vimos antes.

[pcmon.c]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <err.h>
#include "sem.h"

enum {QSIZE = 4};

typedef struct Queue Queue;
struct Queue {
    pthread_mutex_t mutex;
    pthread_cond_t notempty, notfull;
    char buf[QSIZE];
    int hd, tl, sz;
};


static int
qinit(Queue *q)
{
    memset(q, 0, sizeof *q);
    if (pthread_mutex_init(&q->mutex, NULL) != 0) {
        return -1;
    }
    if (pthread_cond_init(&q->notfull, NULL) < 0) {
        pthread_mutex_destroy(&q->mutex);
        return -1;
    }
    if (pthread_cond_init(&q->notempty, NULL) < 0) {
        pthread_cond_destroy(&q->notfull);
        pthread_mutex_destroy(&q->mutex);
        return -1;
    }
    return 0;
}
```

```
static void
qterm(Queue *q)
{
    pthread_cond_destroy(&q->notfull);
    pthread_cond_destroy(&q->notempty);
    pthread_mutex_destroy(&q->mutex);
    q->hd = q->tl = 0;
}


static void
qput(Queue *q, int c)
{
    if (pthread_mutex_lock(&q->mutex) != 0) {
        err(1, "mutex");
    }

    while (q->sz == QSIZE) {
        if (pthread_cond_wait(&q->notfull, &q->mutex) != 0) {
            err(1, "cond wait");
        }
    }

    q->buf[q->tl] = c;
    q->tl = (q->tl+1)%QSIZE;
    q->sz++;

    if (pthread_cond_signal(&q->notempty) != 0) {
        err(1, "cond signal");
    }
    if (pthread_mutex_unlock(&q->mutex) != 0) {
        err(1, "mutex");
    }
}
```

(c) 2016 Francisco J. Ballesteros

```
static int
qget(Queue *q)
{
    int c;

    if (pthread_mutex_lock(&q->mutex) != 0) {
        err(1, "mutex");
    }

    while (q->sz == 0) {
        if (pthread_cond_wait(&q->notempty, &q->mutex) != 0) {
            err(1, "cond wait");
        }
    }

    c = q->buf[q->hd];
    q->hd = (q->hd+1)%QSIZE;
    q->sz--;

    if (pthread_cond_signal(&q->notfull) != 0) {
        err(1, "cond signal");
    }
    if (pthread_mutex_unlock(&q->mutex) != 0) {
        err(1, "mutex");
    }
    return c;
}


static Queue q;


static void*
tput(void *a)
{
    char *s;
    int i;

    s = a;
    for (i = 0; i < 10; i++) {
        qput(&q, s[0]);
    }
    return NULL;
}
```

(c) 2016 Francisco J. Ballesteros

```
static void*
tget(void *a)
{
    int c;
    char buf[1];

    for (;;) {
        c = qget(&q);
        if (c == 0) {
            break;
        }
        if (c < 0) {
            err(1, "qget");
        }
        buf[0] = c;
        if (write(1, buf, 1) != 1) {
            err(1, "write");
        }
    }
    return NULL;
}


int
main(int argc, char* argv[])
{
    pthread_t p1, p2, g;
    void *sts;

    if (qinit(&q) < 0) {
        err(1, "qinit");
    }
    if (pthread_create(&p1, NULL, tput, "a") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&p2, NULL, tput, "b") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&g, NULL, tget, NULL) != 0) {
        err(1, "thread");
    }
    pthread_join(p1, &sts);
    pthread_join(p2, &sts);
    qput(&q, 0);
    pthread_join(g, &sts);
    write(1, "\n", 1);
    qterm(&q);
    exit(0);
}
```

Si lo ejecutamos, veremos que funciona de un modo similar a nuestra última implementación.

(c) 2016 Francisco J. Ballesteros

```
unix$ pcmon
bababaaababaabbbaabb
unix$
```

Te resultará útil en este punto comparar la implementación con semáforos con la implementación con un monitor.

(c) 2016 Francisco J. Ballesteros

Capítulo 8: La red

(c) 2016 Francisco J. Ballesteros

1. Sockets

La red no existía cuando hicieron UNIX. Cuando posteriormente las máquinas empezaron a interconectarse (antes de la llegada de Skynet) nuevas abstracciones aparecieron para poder usar la red en UNIX. Igual sucedió con los terminales gráficos.

El desmadre que puede apreciarse en los interfaces (y en gran parte de las implementaciones) para usar tanto la red como los gráficos en UNIX se debe en parte a que ni Ken Thompson ni Dennis Ritchie estaban ahí cuando los añadieron a UNIX. Mientras tanto, en Bell Labs añadieron interfaces mas razonables para ambas cosas (primero en su versión de UNIX y luego en Plan), pero por desgracia las versiones de los otros se extendieron antes y tenemos que vivir con ello.

La principal abstracción para utilizar la red se conoce como **socket**. Es como un descriptor de fichero (por un extremo) y el otro extremo es la red, con lo que para nosotros un socket es como un calcetín (un tubo con un sólo extemo). De ahí el nombre.

Una vez tenemos creado y configurado un socket podemos utilizarlo para leer y escribir. Siendo un socket, cuando leemos recibimos datos de la red y cuando escribimos enviamos datos a la red. Si utilizamos un protocolo orientado a conexión entonces puede que varios writes se envíen como un sólo mensaje de red, o puede que parte de un write. Si utilizamos un protocolo orientado a datagramas entonces cada write envía un datagrama. Esta es la idea. El interfaz es algo más complicado, aunque no tanto.

Los sockets pueden utilizarse tanto para TCP, como para UDP, como para otros protocolos que no son TCP/IP (que no son de internet). De hecho, existen los llamados *unix domain sockets* que funcionan sólo dentro de una máquina. En la actualidad lo mas práctico es utilizar siempre TCP/IP en lugar de cualquier otro tipo de sockets.

2. Un cliente

Veamos cómo programar un **cliente**, llamado así puesto que hace peticiones a un **servidor**, llamado así puesto que sirve las peticiones de un cliente. Lo que debemos hacer es:

- Crear un socket, en este caso para usar TCP, luego
- construir la dirección a que queremos llamar, y por último
- establecer la conexión a dicha dirección.

Una vez tengamos esto hecho, obtendremos un descriptor de fichero conectado al servidor y podremos usar `read` y `write` para hablar con el otro extremo de la red. Casi como si fuese un pipe full-duplex (bidireccional).

Vamos a implementar una función llamada `dial` a la que daremos un string con la dirección del servidor y un entero con el puerto en el servidor al que queremos conectarnos. Aunque ya has estudiado redes de ordenadores, recordamos que los puertos sirven para hablar con procesos dentro de una máquina y son enteros que no difieren mucho de un apartado de correos.

```
int
dial(char *host, int port)
{
    struct hostent *hent;
    int sfd;
    struct sockaddr_in addr;
```

(c) 2016 Francisco J. Ballesteros

```
        hent = gethostbyname(host);
        if (hent == NULL) {
            return -1;
        }
        memset(&addr, 0, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);
        memmove(&addr.sin_addr.s_addr, hent->h_addr, hent->h_length);



        sfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sfd < 0) {
            return -1;
        }
        fprintf(stderr, "dialing %s:%d...\n", host, port);
        if (connect(sfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
            close(sfd);
            return -1;
        }
        return sfd;
    }
```

Todo el código al principio de la función llama a `gethostbyname` para obtener una dirección IP para el nombre del host al que queremos conectarnos y construye una dirección en la variable `addr`. Ten en cuenta que los sockets usan redes dispares y las direcciones deben soportar las distintas redes. En Plan 9 utilizan strings como tipo de datos, lo que resulta más natural, pero en UNIX es preciso hacer conversiones de tipo y mover los bytes casi a mano. Cuidado así mismo con poner los bytes en el formato de red, como recordarás de tus estudios de redes de ordenadores.

El socket está dentro del kernel. En esta función `sfd` es el descriptor del socket y eso es todo lo que necesitamos. Lo creamos llamando a

```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```

para crear un socket de TCP/IP (`AF_INET`) que hable TCP (`SOCK_STREAM`).

Una vez creado podemos llamar a `connect` para conectarnos a la dirección que pasamos como argumento. En ese momento, nuestro socket abstrae toda la pila de TCP/IP que tiene debajo para hablar con el servidor en el otro extremo de la red.

Con este código, es posible escribir un cliente para, por ejemplo, conectarse a un servidor, escribir una línea de comandos y leer la salida de la ejecución de dicho comando en el servidor.

(c) 2016 Francisco J. Ballesteros

[cli.c]:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <err.h>

int
dial(char *host, int port)
{
    ...como se muestra antes...
}


int
main(int argc, char *argv[])
{
    int fd, port, nr;
    char buf[16*1024];

    if (argc != 4) {
        fprintf(stderr, "usage: %s addr port cmd\n", argv[0]);
        exit(1);
    }
    port = atoi(argv[2]);


    fd = dial(argv[1], port);
    if (fd < 0) {
        err(1, "dial %s:%d", argv[1], port);
    }
    snprintf(buf, sizeof buf, "%s\n", argv[3]);
    buf[sizeof buf - 1] = 0;
    if (write(fd, buf, strlen(buf)) != strlen(buf)) {
        close(fd);
        err(1, "write");
    }
```

(c) 2016 Francisco J. Ballesteros

```
        for(;;) {
            nr = read(fd, buf, sizeof buf-1);
            if (nr < 0) {
                close(fd);
                err(1, "read");
            }
            if (nr == 0) {
                break;
            }
            if (write(1, buf, nr) != nr) {
                close(fd);
                err(1, "write stdout");
            }
        }
        close(fd);
    }
```

3. Un servidor

Para programar un servidor necesitamos poder atender llamadas procedentes de clientes, lo que podemos hacer con un socket. Este socket es preciso configurarlo con la dirección en que queremos escuchar (que incluye el puerto en que escuchamos). Dicho socket se queda escuchando hasta que un cliente se conecta. Además, cada llamada que se acepta se procesa en otro socket distinto para permitir que el socket original continúe escuchando. La siguiente función devuelve un socket que está escuchando en el puerto de TCP que se indica (en cualquier dirección que posea la máquina para dicho protocolo).

```
int
listentcp(int port)
{
    int sfd;
    struct sockaddr_in addr;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);

    sfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sfd < 0) {
        return -1;
    }
    if (bind(sfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        close(sfd);
        return -1;
    }
    if (listen(sfd, 30) < 0) {
        close(sfd);
        return -1;
    }
    return sfd;
}
```

Al principio construimos una dirección para configurar el socket, que guarda el puerto en que queremos escuchar. Usamos una dirección IP que indica que nos vale cualquier dirección IP, lo que en realidad quiere

(c) 2016 Francisco J. Ballesteros

decir que queremos escuchar en todas las que tenga el sistema.

Posteriormente creamos el socket como en el código del cliente, pero, esta vez hemos de llamar a `bind` para darle una dirección al socket. En este punto podemos llamar a `listen` para ponerlo realmente a escuchar. Tras la llamada, UNIX acepta peticiones de conexión (realmente las recibe sin rechazarlas) y mantiene un máximo de 30 en cola (según indica el segundo argumento) mientras nuestro programa se toma su tiempo para llamar a `accept`.

Con esta llamada, tenemos el socket escuchando pero no estamos realmente aceptando peticiones de conexión. Esto es, estamos escuchando pero no *descolgando* el teléfono. La siguiente función "descuelga" cuando llega una llamada pero aceptándola en otra "extensión telefónica" (en otro socket) para que nuestro socket pueda seguir escuchando llamadas.

```
int
acceptcall(int listenfd, char caddr[], int ncaddr)
{
    int fd, port;
    struct sockaddr_in addr;
    socklen_t alen;
    char *cs;


    memset(&addr, 0, sizeof(addr));
    alen = sizeof(addr);
    fd = accept(listenfd, (struct sockaddr*)&addr, &alen);
    if (fd < 0) {
        return -1;
    }
    cs = inet_ntoa(addr.sin_addr);
    port = ntohs(addr.sin_port);
    snprintf(caddr, ncaddr, "%s#%d", cs, port);
    caddr[ncaddr-1] = 0;
    return fd;
}
```

La llamada a `accept` devuelve *otro* socket, cuyo descriptor es *fd*, tras esperar una llamada en el socket `listenfd` que procede del *listen*, y aceptarla. Además, dicha llamada deja en `addr` la dirección del cliente que se ha conectado, lo que aprovechamos para hacer que la función deje en `caddr[]` un string con una cadena que corresponda a dicha dirección.

Ya tenemos todo lo necesario para programar el servidor. Ahora basta ponerlo a escuchar y, para cada llamada, ejecutar un shell que ejecute el comando y escriba la respuesta desde el servidor.

(c) 2016 Francisco J. Ballesteros

[xsrv.c]:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <netdb.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <err.h>

int
listentcp(int port)
{
    ...como se muestra antes...
}
int
acceptcall(int listenfd, char caddr[], int ncaddr)
{
    ...como se muestra antes...
}

int
main(int argc, char *argv[])
{
    int lfd, cfd, infd, port, nr;
    char buf[16*1024];

    if (argc != 2) {
        fprintf(stderr, "usage: %s port\n", argv[0]);
        exit(1);
    }
    port = atoi(argv[1]);
    lfd = listentcp(port);
    if (lfd < 0) {
        err(1, "listen *:%d", port);
    }


    for(;;) {
        cfd = acceptcall(lfd, buf, sizeof buf);
        fprintf(stderr, "call from %s\n", buf);
        switch(fork()) {
        case -1:
            close(cfd);
            close(lfd);
            err(1, "fork");
            break;
```

(c) 2016 Francisco J. Ballesteros

```
        case 0:
            close(lfd);
            // should really read line by line
            nr = read(cfd, buf, sizeof buf-2);
            if (nr < 0) {
                close(cfd);
                err(1, "read");
            }
            if (nr == 0) {
                close(cfd);
            }
            buf[nr++] = '\n';
            buf[nr] = 0;
            fprintf(stderr, "exec %s", buf);
            infd = open("/dev/null", O_RDONLY);
            if (infd < 0) {
                close(cfd);
                err(1, "open");
            }
            dup2(infd, 0);
            close(infd);
            dup2(cfd, 1);
            dup2(cfd, 2);
            close(cfd);
            execl("/bin/sh", "sh", "-c", buf, NULL);
            exit(1);


        default:
            close(cfd);
        }
    }
    exit(1);
}
```

4. Usando el cliente y el servidor

Podemos ejecutar el servidor en un shell

```
unix$ xsrv 4000
```

Y llamarlo con un cliente ejecutado en otro

```
unix$ cli localhost 4000 ls
dialing localhost:4000...
cli
xsrv
cli.c
cli.o
xsrv.c
unix$ cli localhost 4000 date
dialing localhost:4000...
Wed Aug 31 19:40:05 CEST 2016
unix$
```

(c) 2016 Francisco J. Ballesteros

Esto puede verse en la ventana del servidor mientras tanto...

```
unix$ xsrv 4000
call from 127.0.0.1#62961
exec ls

call from 127.0.0.1#62962
exec date
```

Con lo que hemos visto y el manual tienes todo lo necesario para programar servidores y clientes. Las herramientas que hemos aprendido a utilizar durante el curso, como verás, han resultado útiles para hacer nuestro trabajo.

Además, en el futuro, basta con utilizar `dial`, `listentcp` y `acceptcall` en lugar de tener que utilizar directamente el interfaz de sockets. Acóstumbrate a implementar abstracciones como la que proporcionan estas tres funciones: *los sockets como llamadas de teléfono*. Te resultará cómodo para evitar repetir el trabajo que tienes que hacer una y otra vez.

(c) 2016 Francisco J. Ballesteros

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.

(c) 2016 Francisco J. Ballesteros